Translating Parameter Estimation Problems

From EASY-FIT to SOCS

A Thesis Submitted to the

College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Matthew W. Donaldson

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

Abstract

Mathematical models often involve unknown parameters that must be fit to experimental data. These so-called parameter estimation problems have many applications that may involve differential equations, optimization, and control theory. EASY-FIT and $\mathbb{SOCS}$ are two software packages that solve parameter estimation problems. In this thesis, we discuss the design and implementation of a source-to-source translator called EF2$\mathbb{SOCS}$ used to translate EASY-FIT input into $\mathbb{SOCS}$ input. This makes it possible to test $\mathbb{SOCS}$ on a large number of parameter estimation problems available in the EASY-FIT problem database that vary both in size and difficulty.

Parameter estimation problems typically have many locally optimal solutions, and the solution obtained often depends critically on the initial guess for the solution. A 3-stage approach is followed to enhance the convergence of solutions in $\mathbb{SOCS}$. The stages are designed to use an initial guess that is progressively closer to the optimal solution found by EASY-FIT. Using this approach we run EF2$\mathbb{SOCS}$ on all 691 translatable problems from the EASY-FIT database. We find that all but 7 problems produce converged solutions in $\mathbb{SOCS}$. We describe the reasons that $\mathbb{SOCS}$ was not able solve these problems, compare the solutions found by $\mathbb{SOCS}$ and EASY-FIT, and suggest possible improvements to both EF2$\mathbb{SOCS}$ and $\mathbb{SOCS}$.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Raymond J. Spiteri, whose guidance, patience, enthusiasm, and endless supply of great ideas throughout the course of my entire university education has served to encourage me to successfully reach my goals.

A very special thank you goes to Dr. John T. Betts, who over the years has dedicated his time and effort to unselfishly help me learn the use of Boeing's Sparse Optimal Control Software (SOCS).

Finally, I would like to thank my parents, Gilbert Donaldson and Trudy Donaldson, whose support and encouragement has made this accomplishment possible. I am truly blessed to have such wonderful and caring parents.

Contents

List of Tables

List of Figures

List of Abbreviations

BDF	Backward Differentiation Formula
DAE	Differential-Algebraic Equation
GUI	Graphical User Interface
IVP	Initial-Value Problem
NLP	Nonlinear Programming
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
PSE	Problem-Solving Environment
$\mathbb{SOCS}$	Sparse Optimal Control Software
SQP	Sequential Quadratic Programming

Chapter 1

Introduction

Optimal control problems involve the optimization of some performance index over time subject to a given dynamical system. Optimal control is applied in many fields ranging from industrial engineering and military applications to economics, medicine, and biology. For example, aerospace engineers have been solving optimal control problems for trajectory specification, spacecraft altitude planning, jet thruster operation, missile guidance, and many other applications for decades [5]. In economics, optimal control is applied to the management of resources [27]. Also, the design of effective medical treatments can be formulated as an optimal control problem [30].

The dynamical systems encountered in optimal control problems are often defined mathematically by sets of ordinary differential equations (ODEs). In general, analytical solutions of optimal control problems are only possible for very simple problems. For this reason, we must approximate the solution of optimal control problems numerically. Based on this general description, there are traditionally two major parts of an effective optimal control solution technique. The first is the "discretization" part, and the second is the "optimization" part. Although naive approaches that involve "pasting" together packages for numerical integration and optimization may be moderately successful, there exist better ways [5]. Several numerical software packages exist for the numerical solution of optimal control problems. In this thesis, we describe two such packages in detail, namely, EASY-FIT [15] and $\mathbb{SOCS}$ [7]. We also discuss the design and implementation of a source-to-source translator called EF2$\mathbb{SOCS}$ used to translate EASY-FIT input into $\mathbb{SOCS}$ input.

EASY-FIT is an interactive software system used to identify model parameters in explicit model functions and dynamical systems of equations [15]. These *parameter estimation problems* form a subset of the more general optimal control problem; see Chapter 2. More precisely, parameter

estimation problems involve fitting a mathematical model with unknown parameters to experimental data such that some measure of the error in the fit is minimized. A formal description of these problems and their relationship to optimal control problems can be found in Section 2.2. The mathematical background of the numerical algorithms implemented in EASY-FIT is described in the form of a comprehensive textbook [45]. EASY-FIT is implemented in the form of a relational database under Microsoft Access running under Microsoft Windows (2000 or higher). The underlying numerical algorithms are coded in FORTRAN77. The current version is 4.0.

The industrial software package $\mathbb{SOCS}$ (Sparse Optimal Control Software) is a collection of FORTRAN90 subroutines developed by The Boeing Company that are designed to solve optimal control problems [7]. The package implements the *direct transcription method* to convert the continuous control problem into a discrete one. The discretization gives a finite-dimensional nonlinear programming (NLP) problem, which is solved by $\mathbb{SOCS}$ using, for example, a sequential programming (SQP) method [18]. $\mathbb{SOCS}$ is supported on most UNIX and Windows systems. The current version is 7.0.0.

As valuable information to the developers of $\mathbb{SOCS}$, we would like to use $\mathbb{SOCS}$ to solve a large test set of problems and compare the results to solutions obtained from other software packages. In the process, we hope to uncover potential deficiencies in the software. There are several issues that make this comparison a challenging task. Briefly, three immediate problems we encounter are:

1. locating sources for a significant number of nontrivial parameter estimation problems;

2. spending a substantial amount of time and effort to manually code these problems in $\mathbb{SOCS}$;

3. using a second software package in order to obtain solutions for the purposes of comparison.

Fortunately, with the help of EASY-FIT and the design of the source-to-source translator EF2$\mathbb{SOCS}$, we are able to address all of these problems and simplify the process of evaluating $\mathbb{SOCS}$ for solving parameter estimation problems.

First, we take advantage of the large database of parameter estimation problems in EASY-FIT. There are over 1000 parameter estimation problems that are included in the EASY-FIT database.

They are based on academic and real-life examples. These problems are specified in a FORTRAN77-like modelling language called PCOMP [45]. PCOMP focuses on creating short and efficient code to define the model functions. With the use of a batch file we are easily able to solve any set of these problems using EASY-FIT. Having found a suitable test set of problems and corresponding solutions, we focus on the idea of a translator.

The main motivation behind designing EF2𝕊𝕆ℂ𝕊 is to allow for the translation of EASY-FIT input into 𝕊𝕆ℂ𝕊 input. Specifically, the job of EF2𝕊𝕆ℂ𝕊 is to take a parameter estimation problem specified as a PCOMP script from the EASY-FIT database and output executable 𝕊𝕆ℂ𝕊 code. There are three advantages resulting from the design of EF2𝕊𝕆ℂ𝕊. First, we have access to all the problem descriptions and solutions of the parameter estimation problems in the database of EASY-FIT without the need to consult multiple sources, such as journals, textbooks, and other references. Second, we are able to specify new problems in the high-level modelling language PCOMP. Finally, we can avoid most if not all manual coding of FORTRAN programs to run with 𝕊𝕆ℂ𝕊.

With the main goal of designing EF2𝕊𝕆ℂ𝕊 for source-to-source translation and testing 𝕊𝕆ℂ𝕊 on a large number of parameter estimation problems, the thesis is presented as follows. In Chapter 2, we formally present the optimal control problem and give examples of some numerical software available to solve such problems. We also give details on the problem formulations of parameter estimation problems involving explicit model functions, ODEs, and differential-algebraic equations (DAEs). In Chapter 3, we describe the design of EF2𝕊𝕆ℂ𝕊. We also give a simple example of how to use EF2𝕊𝕆ℂ𝕊 on a parameter estimation problem involving an explicit model function. In Chapter 4, we present the results of running EF2𝕊𝕆ℂ𝕊 on 691 problems from the EASY-FIT database that can be translated and provide some suggestions from improving 𝕊𝕆ℂ𝕊. Finally, in Chapter 5 we discuss the conclusions and future work.

Chapter 2

Parameter Estimation and Optimal Control Problems

In optimal control problems, we seek a control function that yields a trajectory of a dynamical system that simultaneously satisfies equations of motion, boundary conditions, and equality and/or inequality constraints, while optimizing a performance index (or cost function). We begin this chapter with a brief description of the historical development of optimal control problems. We then present a detailed mathematical description of the optimal control problem. We summarize the solution techniques used by $\mathbb{SOCS}$ and EASY-FIT as well as 4 other numerical software packages for optimal control. Parameter estimation problems form a subset of optimal control problems; see Section 2.2. We describe the 3 categories of parameter estimation problems with which we are concerned in this thesis. Finally, we present the basic idea of a translator and its role in solving optimal control problems.

2.1 Optimal Control Problems

Optimal control problems, also referred to as *dynamic optimization problems*, involve the optimization of some performance index over time, subject to a given dynamical system and other constraints. The decision variables of the optimization aspect of the problem are known as the *controls*, and the solution of the dynamical system for a prescribed control is called the *state*. Different types of constraints may be imposed on both the control and the state [9].

Optimal control problems arise in a wide variety of disciplines including different fields of engineering, economics, and biology. Many fundamental results in optimal control theory have been

obtained through the joint efforts of engineers, mathematicians, scientists, and economists. The range of real-world problems that can be solved by optimal control theory is constantly growing [10]. Historically, some interesting optimal control problems have been solved analytically [10]. However, as the problems considered become more and more complex, it is no longer possible to solve them analytically. Mathematicians, engineers, and others are therefore interested in the development of new and more powerful computational methods for solving these problems numerically. In this section, after defining an optimal control problem, we present some of these computational methods and their implementation in the form of software packages.

2.1.1 Historical Development of Optimal Control

According to Bryson [10], optimal control had its origins in the calculus of variations in the 17th century based on the work of Pierre de Fermat (1601–1665), Isaac Newton (1642–1727), Gottfried Wilhelm Liebnitz (1646–1716), and Johann Bernoulli (1667–1748). The calculus of variations was further developed over the following two centuries by Leonard Euler (1707–1783), Jean Louis Lagrange (1736–1813), and William Rowan Hamilton (1805–1865). In 1962, Lev Semenovich Pontryagin (1908-1988) extended the calculus of variations to handle control variable inequality constraints, leading to the *Pontryagin maximum principle*, which gives a necessary condition that must hold on an optimal trajectory [40]. The truly enabling element for the use of optimal control theory was the digital computer, which became commercially available in the 1950s. In the 1980s, research began and continues today on making optimal control more robust to variations in dynamical system and disturbance models [10].

2.1.2 The Optimal Control Problem

As described by [5], an optimal control problem can be formulated as a collection of N phases where the independent variable t for phase K is defined in $t_I^{(K)} \leq t \leq t_F^{(K)}$. A *phase* is defined to be a portion of a trajectory in which the dynamics of the system remain unchanged. Within phase K, the dynamics of the system are described by $n_d^{(K)}$ *dynamic variables* $\mathbf{d}^{(K)}(t)$ made up of $n_y^{(K)}$

state variables $\mathbf{y}^{(K)}(t)$ and $n_u^{(K)}$ control variables $\mathbf{u}^{(K)}(t)$:

$$\mathbf{d}^{(K)}(t) = \begin{bmatrix} \mathbf{y}^{(K)}(t) \\ \mathbf{u}^{(K)}(t) \end{bmatrix}, \tag{2.1}$$

where $\mathbf{y}^{(K)} \in \mathbb{R}^{n_y^{(K)}}$, $\mathbf{u}^{(K)} \in \mathbb{R}^{n_u^{(K)}}$, and $n_d^{(K)} = n_y^{(K)} + n_u^{(K)}$.

Typically, the dynamics of the system are defined by a set of ODEs,

$$\dot{\mathbf{y}}^{(K)} = \mathbf{f}^{(K)}[\mathbf{y}^{(K)}(t), \mathbf{u}^{(K)}(t), t], \tag{2.2}$$

where $\mathbf{f}^{(K)} : \mathbb{R}^{n_y^{(K)}} \times \mathbb{R}^{n_u^{(K)}} \times \mathbb{R} \to \mathbb{R}^{n_y^{(K)}}$. The solution must satisfy algebraic constraints of the form

$$\mathbf{g}_L^{(K)} \leq \mathbf{g}^{(K)}[\mathbf{y}^{(K)}(t), \mathbf{u}^{(K)}(t), t] \leq \mathbf{g}_U^{(K)}, \tag{2.3}$$

where $\mathbf{g} : \mathbb{R}^{n_y^{(K)}} \times \mathbb{R}^{n_u^{(K)}} \times \mathbb{R} \to \mathbb{R}^{n_g^{(K)}}$ is composed of $n_=^{(K)}$ equality constraints and $n_\geq^{(K)}$ inequality constraints, including simple bounds on the state variables

$$\mathbf{y}_L^{(K)} \leq \mathbf{y}^{(K)}(t) \leq \mathbf{y}_U^{(K)} \tag{2.4}$$

and control variables

$$\mathbf{u}_L^{(K)} \leq \mathbf{u}^{(K)}(t) \leq \mathbf{u}_U^{(K)}. \tag{2.5}$$

The phases are linked by boundary conditions of the form

$$\begin{aligned} \boldsymbol{\psi}_L \leq \ \boldsymbol{\psi} \Big[& \mathbf{y}^{(1)}(t_I^{(1)}), \mathbf{u}^{(1)}(t_I^{(1)}), t_I^{(1)}, \\ & \mathbf{y}^{(1)}(t_F^{(1)}), \mathbf{u}^{(1)}(t_F^{(1)}), t_F^{(1)}, \\ & \ldots, \\ & \mathbf{y}^{(N)}(t_I^{(N)}), \mathbf{u}^{(N)}(t_I^{(N)}), t_I^{(N)}, \\ & \mathbf{y}^{(N)}(t_F^{(N)}), \mathbf{u}^{(N)}(t_F^{(N)}), t_F^{(N)} \Big] \ \leq \boldsymbol{\psi}_U, \end{aligned} \tag{2.6}$$

where $\boldsymbol{\psi}_L$ and $\boldsymbol{\psi}_U$ are vectors of appropriate size. These boundary conditions allow the values of the dynamic variables at the beginning and/or end of any phase to be related to those of any other phase.

The optimal control problem is to determine the n_u-dimensional control vector $\mathbf{u}(t)$, consisting of the concatenation in time of the controls $u^{(K)}(t),\ K = 1, 2, \ldots, N$, on each phase that minimizes

the performance index

$$
\begin{aligned}
J = \phi \Big[& \mathbf{y}^{(1)}(t_I^{(1)}), \mathbf{u}^{(1)}(t_I^{(1)}), t_I^{(1)}, \\
& \mathbf{y}^{(1)}(t_F^{(1)}), \mathbf{u}^{(1)}(t_F^{(1)}), t_F^{(1)}, \\
& \ldots, \\
& \mathbf{y}^{(N)}(t_I^{(N)}), \mathbf{u}^{(N)}(t_I^{(N)}), t_I^{(N)}, \\
& \mathbf{y}^{(N)}(t_F^{(N)}), \mathbf{u}^{(N)}(t_F^{(N)}), t_F^{(N)} \Big] \\
& + \sum_{K=1}^{N} \left\{ \int_{t_I^{(K)}}^{t_F^{(K)}} \mathbf{q}^{(K)} \left[\mathbf{y}^{(K)}(t), \mathbf{u}^{(K)}(t), t \right] \, dt \right\},
\end{aligned}
\tag{2.7}
$$

that may involve *quadrature functions* $\mathbf{q}^{(K)}$. Equations (2.1)–(2.7) specify a general optimal control problem with multiple phases, multiple states, two-point boundary conditions, and a general form for the performance index. In order to solve the optimal control problem, it is often convenient to make use of the necessary conditions of optimality. We now give the necessary conditions for a single-phase problem, where the initial conditions for the state variables are given, and the performance index J does not involve quadrature functions. For clarity, we omit the phase-dependent notation from the remaining discussion. We define an augmented performance index

$$
\hat{J} = [\phi + \boldsymbol{\nu}^{\mathsf{T}} \boldsymbol{\psi}]_{t_F} - \int_{t_I}^{t_F} \boldsymbol{\lambda}^{\mathsf{T}} \{ \dot{\mathbf{y}} - \mathbf{f}[\mathbf{y}(t), \mathbf{u}(t), t] \} \, dt, \tag{2.8}
$$

where $\boldsymbol{\nu} \in \mathbb{R}^{\dim(\boldsymbol{\psi})}$ represents the Lagrange multipliers for the pointwise constraints (2.6) and $\boldsymbol{\lambda}(t) \in \mathbb{R}^{n_y}$ represents multipliers that we call *adjoint* or *co-state variables* for the continuous constraint (2.2). The necessary conditions for a constrained optimum are obtained by setting the first variation $\delta \hat{J} = 0$ [9]. It is convenient to define the *Hamiltonian*

$$
H = \boldsymbol{\lambda}^{\mathsf{T}} \mathbf{f}[\mathbf{y}(t), \mathbf{u}(t), t] \tag{2.9}
$$

and the auxiliary function

$$
\Phi = \phi + \boldsymbol{\nu}^{\mathsf{T}} \boldsymbol{\psi}. \tag{2.10}
$$

The necessary conditions, called the *Euler–Lagrange equations*, which result from setting $\delta \hat{J} = 0$, in addition to (2.2) and (2.6), are

$$
\dot{\boldsymbol{\lambda}} = -\mathbf{H}_{\mathbf{y}}{}^{\mathsf{T}}, \tag{2.11}
$$

called the *adjoint equations*,

$$\mathbf{0} = \mathbf{H_u}^{\mathsf{T}}, \tag{2.12}$$

called the *control equations*, and

$$\boldsymbol{\lambda}(t_F) = \mathbf{\Phi_y}^{\mathsf{T}}\Big|_{t=t_F}, \tag{2.13}$$

$$0 = (\Phi_t + H)\big|_{t=t_F}, \tag{2.14}$$

called the *transversality conditions*. In the above equations, $\mathbf{0} \in \mathbb{R}^{n_y}$ and the partial derivatives $\mathbf{H_y}$, $\mathbf{H_u}$, and $\mathbf{\Phi_y}$ are row vectors; e.g., $\mathbf{H_y} \equiv (\partial H/\partial y_1, \ldots, \partial H/\partial y_{n_y})$. The control equations (2.12) are a simplified statement of the *Pontryagin maximum principle* [40]. The Pontryagin maximum principle states that the control variable must be chosen to optimize the Hamiltonian at every point in time subject to limitations on the control imposed by state and control constraints. The complete set of necessary conditions consists of a DAE system (2.2), (2.11), and (2.12) with boundary conditions at t_I and t_F (2.6), (2.13), and (2.14). This is often referred to as a *two-point boundary value problem*. A more in-depth treatment of this material can be found in [9].

2.1.3 Direct versus Indirect Solution Methods

In practice, the methods for solving optimal control problems are classified as either *direct* or *indirect*. Indirect methods proceed by formulating the optimality conditions (2.2), (2.6), (2.11), (2.12), (2.13), and (2.14), according to the Pontryagin maximum principle. The resulting two-point boundary value problem is then solved numerically. Therefore, indirect methods use an "optimize then discretize" technique [6]. On the other hand, direct methods do not require explicit derivation and construction of the necessary conditions [5]; i.e., a direct method does not construct the adjoint equations (2.11), the control equations (2.12), nor either of the transversality (boundary) conditions (2.13)–(2.14). Direct methods discretize the optimal control problem in time and solve the resulting finite-dimensional parameter optimization problem. Therefore, direct methods generate an approximate solution to the original problem as opposed to the necessary conditions. These methods use a "discretize then optimize" technique [6]. The differences between direct and indirect methods are considered in [43].

Indirect methods yield highly accurate results; however, there are some drawbacks. First, the user must compute the quantities $\mathbf{H_y}$, $\mathbf{H_u}$, etc., that appear in the defining equations (2.11)–(2.14). Evaluating these equations may be computationally expensive. Second, the user must find a sufficiently good guess for the values of the adjoint variables $\boldsymbol{\lambda}(t)$ in order to achieve convergence to the solution. Because the adjoints are nonphysical quantities, finding an effective initial guess is often nonintuitive. Even with a reasonable guess for the adjoint variables, the numerical solution of the adjoint equations can be ill-conditioned. Therefore, indirect methods may be very sensitive to small changes in the unspecified boundary conditions. It is not unusual for the numerical integration, with poorly guessed initial conditions, to produce "wild" trajectories in the state space [9].

Direct methods are often applied in industrial optimization problems, where faster and more robust methods are required with modest accuracy requirements. A key advantage of direct methods over indirect methods is that they do not require a user-supplied estimate of the adjoint variables. Therefore, less preparatory work is required from the user in order to use these methods. Moreover, they generally have a larger domain of convergence. However, the trade off in comparison to indirect methods is that it is computationally more expensive to get a high-accuracy solution using direct methods compared to indirect methods because direct methods approximate the solution of the original problem.

Because direct and indirect methods both have their relative advantages and disadvantages, it is possible to combine them in a hybrid approach. This possibility was examined in a two-stage approach proposed by Bulirsch, Nerz, Pesch, and von Stryk [11].

2.1.4 Numerical Software for Optimal Control

In general, a closed-form solution of an optimal control problem exists only for simple problems. Accordingly, we must approximate the solutions to optimal control problems numerically. We begin by giving a brief description of the two software packages $\mathbb{SOCS}$ [7] and EASY-FIT [15] considered in this thesis. We follow this with descriptions of 4 other existing software packages for numerically solving optimal control problems, i.e., COOPT [53], RIOTS_95 [51], DIRCOL [58], and

MISER3 [29].

The Sparse Optimal Control Software $\mathbb{SOCS}$ is a collection of FORTRAN90 subroutines developed by The Boeing Company that are capable of solving nonlinear optimization, optimal control, and parameter estimation problems with equality and inequality constraints, as well as providing numerical estimates for the required function derivatives. The package implements the *direct transcription* or *collocation* method to discretize the continuous control problem. The discretization yields a finite-dimensional sparse NLP problem that can be solved using a sparse optimization algorithm such as SQP or interior point algorithm [18]. Numerical procedures to improve the accuracy of the discretization by mesh refinement are implemented in the package. Details of the various subroutines used by $\mathbb{SOCS}$ are found in Chapter 3.1. Some of the current listings of applications solved by $\mathbb{SOCS}$ that are given on the company website include low thrust orbit transfer, golf putting, and interplanetary orbit transfer [38].

The EASY-FIT software package is an interactive software system used to identify model parameters and implemented in the form of a relational database under Microsoft Access running under Microsoft Windows (2000 or higher). The underlying numerical algorithms are coded in FORTRAN77 and are executable independently from the interface. Model functions are either interpreted and evaluated symbolically by user-provided FORTRAN77 subroutines or by a programming language called PCOMP that permits automatic differentiation of nonlinear model functions. Least-squares subroutines are executed to solve the parameter estimation problems. The subroutine DFNLP transforms the original problem into a general NLP problem such that typical features of a Gauss–Newton and quasi-Newton least-squares method are retained. DFNLP is able to take linear or nonlinear constraints into account. The resulting optimization problem is solved by a standard SQP code called NLPQL [44]. In [45], EASY-FIT solves several case studies representing typical application problems in biochemistry, chemistry, pharmaceutics, and electrical, mechanical, and chemical engineering.

The COOPT software package implements a direct method with modified multiple shooting [32] techniques for solving optimal control problems of large-scale DAE systems. The basic approach

in COOPT is to divide the original time interval into multiple shooting intervals, with the DAEs solved numerically on the subintervals at each optimization iteration. State and control continuity constraints are imposed across the subintervals. The resulting optimization problem is solved by sparse SQP methods. Partial derivative matrices needed for the optimization are generated by the DAE sensitivity software DASPK3.0 [35]. When integrating on each subinterval, DASPK3.0 requires the computation of the residual of the state and sensitivity equations as well as a computation of the Jacobian of the model equations with respect to the states and their derivatives. The residual measures the difference between the measured and predicted values. The sensitivity equations are used to obtain the values and Jacobians of the nonlinear state continuity constraints enforced between subintervals. The sensitivity equations to be solved are generated via automatic differentiation using ADIFOR [8]. Although these derivatives may use large amounts of computer memory when compared to alternative methods such as finite difference derivatives, ADIFOR derivatives are exact (to within round-off errors) [56]. COOPT has been successfully used to solve optimal control problems arising in a wide variety applications, such as chemical vapour deposition of superconducting thin films [41], spacecraft trajectory design and contingency/recovery problems [52], and computation of cell traction forces in tissue engineering [60].

The RIOTS_95 software package is a group of programs and utilities written mostly in C and designed as a toolbox for the Matlab problem-solving environment, thus providing an interactive environment for solving a broad class of optimal control problems. All of the functionality of Matlab, including command-line execution, data entry, and data plotting, are thus available to the RIOTS_95 user. The numerical methods used by RIOTS_95 are supported by the theory in [48]–[50]; this theory uses the approach of *consistent approximations* as defined by [49]. In this approach, a solution is obtained as an accumulation point of the solutions to a sequence of discrete-time optimal control problems that are consistent approximations to the original continuous-time optimal control problem. The discrete-time optimal control problems are constructed by discretizing the system dynamics with one of 4 fixed-step-size Runge–Kutta integration methods and by representing the controls as finite-dimensional B-splines. This allows for a high degree of function approximation

accuracy without requiring a large number of control parameters. The integration proceeds on a (non-uniform) mesh that specifies the spline breakpoints. The solution obtained for one discretized problem can be used to select a new integration mesh, on which the optimal control problem can be re-discretized to produce a new discrete-time problem that more accurately approximates the original problem. In practice, only a few such re-discretizations are needed to achieve an acceptable solution. RIOTS_95 may be used to solve various optimal control problems such as the classic Goddard rocket problem [54] as well as many others [51].

The DIRCOL software package is a set of FORTRAN77 subroutines designed to solve optimal control problems. Discontinuities in the right-hand side of the differential equations can be treated. DIRCOL implements a direct collocation method [57]. By a discretization of state and control variables, the infinite-dimensional optimal control problem is transcribed into a sequence of (finite-dimensional) nonlinearly constrained optimization problems. Optimal control theory and adjoint differential equations are not required in order to apply the algorithm. The NLP problems are solved either by the dense SQP method NPSOL [23] or by the sparse SQP method SNOPT [22]. DIRCOL also computes reliable estimates of the adjoint variables and the multiplier functions of state constraints by direct discretization of (2.11) and (2.12). Therefore the method can conveniently be combined with an indirect method such as multiple shooting [59]. Supplementary programs are provided for supporting a visualization of the numerical results. For a list of applications that use DIRCOL, including robotics, flight mechanics, biomechanics, automotive engineering, and economics, see [58].

The MISER3 software package is a suite of FORTRAN77 programs for solving continuous and discrete-time optimal control problems subject to general constraints. Discrete-time optimal control problems involve discretizing the state equations (2.2), forming a vector of NLP variables composed of the states and controls evaluated at a finite number of points in time. Included in the software package is a program called DMISER3 for solving discrete-time optimal control problems. This program is almost identical to MISER3 except that it applies to problems in which the dynamics are described by difference equations. The companion program MISER3 is used for solving continuous-

time problems. The method used is based on the idea of *control parametrization*, where the controls are approximated by piecewise constant or piecewise linear functions defined on suitable partitions of the time interval. The code then converts the problem into an NLP problem that is solved using an SQP algorithm. For this purpose, MISER3 contains a state-of-the-art SQP algorithm provided by [44]. For a discussion of the concept of control parametrization as implemented in MISER3, see [28]. According to [17], DMISER3 has been tested on a wide range of problems. A discretization of the problem of a buckled beam with internal loading is described in detail because it includes a wide variety of types of constraints [17], [24].

Solving optimal control problems numerically can be challenging. Often very large parameter bounds can result in solutions that are only locally optimal. Therefore it may be desirable to use global optimization methods. However, if rigorous proofs of global optimality are also desired, the added computational expense associated with these methods generally makes them infeasible except on small problems [16].

Global optimization strategies can be sorted into two general groups: deterministic and stochastic [36]. Although deterministic methods provide some guarantee of convergence, the associated computational cost typically increases rapidly with problem size. On the other hand, stochastic methods rely on some degree of randomness to approach the global optimum. Because of this randomness there is no guarantee of ever reaching the global optimum. The major challenge in solving large-scale global optimization problems is that there is no mathematical basis for efficiently reaching a global minimizer, such as there is for Newton's method for reaching a local minimizer. Many methods have been developed for global optimization problems, but these tend to be heuristic in nature and often require large amounts of computation time [39].

2.2 Parameter Estimation Problems

Parameter estimation is very important in areas such as natural science, engineering, and many other disciplines. The key idea is to estimate unknown parameters $\mathbf{p} = (p_1, \ldots, p_{n_u})^T$ of a mathematical model that often describes a real life situation by minimizing the distance of known

experimental data from theoretically predicted values of a model function at certain time values. In other words, model parameters that cannot be measured directly, or with sufficient precision, are estimated by a least-squares fit [45]. The difference between this problem and the optimal control problem involves a different performance index; i.e., the least-squares performance index is a special case of the more general performance index given by (2.8). The solution of an optimal control problem approximates an infinite-dimensional problem whereas the solution to a parameter estimation problem approximates a finite-dimensional problem. Also, in the optimal control problem (2.2)–(2.7), the control variables are generally time-dependent; in the parameter estimation problem they must be constant.

Formally, we want to solve a least-squares problem of the form

$$\begin{aligned} \min_{\mathbf{p}} \ & \sum_{k=1}^{m_{exp}} \sum_{i=1}^{m_t} \sum_{j=1}^{m_c} (w_{ij}^k (h_k(\mathbf{p}; \mathbf{y}(\mathbf{p}; t_i, c_j), t_i, c_j) - \hat{h}_{ij}^k))^2, \\ & \mathbf{p}_L \leq \mathbf{p} \leq \mathbf{p}_U \ , \ \mathbf{p} \in \mathbb{R}^{n_u}, \end{aligned} \tag{2.15}$$

where $\mathbf{h}(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c), t, c)$ is a fitting function depending on the unknown parameter vector $\mathbf{p}$, the time t, the concentration variable c, and the solution $\mathbf{y}(\mathbf{p}; t, c)$ of a dynamical system. There are m_{exp} experimental data sets with m_t time values for m_c concentration values. The experimental values at time t_i, concentration c_j, and experimental data set h_k are denoted by $\hat{h}_{ij}^k$. We assume that there are weight factors $w_{ij}^k \geq 0$ given by the user that reflect the individual influence of a measurement value on the fit. Weights may be set to zero if for example there are no measurements available for a particular time value. The inequalities on the parameter vector $\mathbf{p}$ are understood to hold componentwise. A typical dynamical system is given by differential equations that describe a time-dependent process and that depend on the parameter vector $\mathbf{p}$. In this section, we summarize in detail how the model functions depend on the solution of a dynamical system. Moreover, we describe several extensions of the data-fitting problem and the dynamical system that allow us to treat more complex practical models.

In general, parameter estimation problems are often classified using one the following categories [45],

- explicit model functions,

- Laplace transformations,
- systems of ODEs,
- systems of DAEs,
- systems of time-dependent partial differential equations (PDEs),
- systems of partial differential-algebraic equations (PDAEs).

These are the categories of problems that can be found in the EASY-FIT database. Of these 6 categories, EF2$\mathbb{SOCS}$ is designed to deal only with explicit model functions, systems of ODEs, and systems of DAEs. However, by using a technique called the *method of lines*, the spatial derivatives of a system of PDEs can be discretized, resulting in a system of ODEs [2]. This technique may also be used to transform a system of PDAEs into DAEs. Therefore, the user may derive these equations and use EF2$\mathbb{SOCS}$ to solve the resulting system of ODEs or DAEs. We now take a more in-depth look at the three categories listed above that can be handled by EF2$\mathbb{SOCS}$.

2.2.1 Explicit Model Functions

Problems where the vector-valued model function $\mathbf{h}(\mathbf{p}; t, c)$ in (2.15) is available in explicit form belong to the class of parameter estimation problems known as *explicit model functions*. Associated with explicit models is an additional variable called *time*, t, and optionally another variable called *concentration*, c, that represents *known* parameters in a given experiment. These names reflect a common usage in practical situations. The terms may have other meanings depending on the underlying application.

As described by [45], we begin with m_{exp} experimental data sets

$$(t_i, c_j, \hat{h}_{ij}^k),\ i = 1, \ldots, m_t,\ j = 1, \ldots, m_c,\ k = 1, \ldots, m_{exp}, \tag{2.16}$$

where m_t time values, m_c concentration values, and $M = m_t m_c m_{exp}$ corresponding measurement values are given. We may also have nonlinear restrictions in the form of equality or inequality constraints, depending on the parameter vector $\mathbf{p}$ to be estimated, and certain time and concentration

values t and c, respectively,

$$\begin{aligned} \mathbf{g}_{=}(\mathbf{p};t,c) &= \mathbf{0}, \\ \mathbf{g}_{\geq}(\mathbf{p};t,c) &\geq \mathbf{0}, \end{aligned} \tag{2.17}$$

where $\mathbf{g}_{=} : \mathbb{R}^{n_u} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{n_=}$ and $\mathbf{g}_{\geq} : \mathbb{R}^{n_u} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{n_\geq}$. We assume that all constraint functions are continuously differentiable with respect to $\mathbf{p}$. Otherwise, this may lead to an ill-posed problem, a consequence of which is that arbitrarily small changes in the data may lead to arbitrarily large changes in the solution.

Together with a vector-valued model function

$$\mathbf{h}(\mathbf{p};t,c) = (h_1(\mathbf{p};t,c),\ldots,h_{m_{exp}}(\mathbf{p};t,c))^T, \tag{2.18}$$

we get the resulting least-squares problem

$$\begin{aligned} \min_{\mathbf{p}} \; & \sum_{k=1}^{m_{exp}} \sum_{i=1}^{m_t} \sum_{j=1}^{m_c} (w_{ij}^k (h_k(\mathbf{p};t_i,c_j) - \hat{h}_{ij}^k))^2, \\ & \mathbf{g}_{=}(\mathbf{p};t_i,c_j) = \mathbf{0}, \\ & \mathbf{g}_{\geq}(\mathbf{p};t_i,c_j) \geq \mathbf{0}, \\ & \mathbf{p}_L \leq \mathbf{p} \leq \mathbf{p}_U, \; \mathbf{p} \in \mathbb{R}^{n_u}. \end{aligned} \tag{2.19}$$

We wish to minimize the norm of the difference between measured experimental data and theoretically predicted values of a model function at certain time and concentration values. This difference is called the *residual*, which is a vector with components denoted by $r_{ij}^k,\ i = 1,\ldots,m_t,\ j = 1,\ldots,m_c,\ k = 1,\ldots,m_{exp}$. Note that for each of the three types of parameter estimation problems described in this Section it is possible in EASY-FIT to have a global scaling strategy in addition to the individual weight factors for each measurement value. The four options are:

- 0: no additional scaling
- 1: division of residuals by square root of sum of squares of measurement values for each data set, i.e.,

$$\frac{r_{ij}^k}{\sqrt{\sum_{i=1}^{m_t} \sum_{j=1}^{m_c} (\hat{h}_{ij}^k)^2}} \tag{2.20}$$

- -1: division of each single residual by corresponding absolute measurement value, i.e.,

$$\frac{r_{ij}^k}{|\hat{h}_{ij}^k|} \tag{2.21}$$

- -2: division of each single residual by corresponding squared measurement value, i.e.,

$$\frac{r_{ij}^k}{[\hat{h}_{ij}^k]^2} \tag{2.22}$$

2.2.2 ODEs

As with the explicit data-fitting model and as described by [45], we begin with m_{exp} data sets,

$$(t_i, c_j, \hat{h}_{ij}^k),\ i = 1, \ldots, m_t,\ j = 1, \ldots, m_c,\ k = 1, \ldots, m_{exp}, \tag{2.23}$$

where m_t time values, m_c concentration values, and $M = m_t m_c m_{exp}$ corresponding measurement values are given. We may also have nonlinear restrictions in the form of inequality constraints,

$$\mathbf{g}_{\geq}(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c)) \geq \mathbf{0}, \tag{2.24}$$

where $\mathbf{y}(\mathbf{p}; t, c)$ represents the solution of the system of ODEs (2.2), depending on the parameter vector $\mathbf{p}$ to be estimated and certain time and concentration values t and c, respectively.

We assume that all constraint functions are continuously differentiable with respect to $\mathbf{p}$. Note that dynamical constraints should be defined only in the form of inequalities; equality constraints are to be treated as algebraic equations that are part of the DAE model; see Section 2.2.3.

The vector-valued model function

$$\mathbf{h}(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c), t, c) = (h_1(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c), t, c), \ldots, h_{m_{exp}}(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c), t, c))^T, \tag{2.25}$$

depends on the concentration parameter c and in addition on the solution $\mathbf{y}(\mathbf{p}; t, c)$ of a system of n_y ODEs with initial values,

$$\begin{aligned} \dot{y}_1 &= f_1(\mathbf{p}; y, t, c)\ , \quad y_1(t_I) = y_1^I(\mathbf{p}; c), \\ &\vdots \\ \dot{y}_{n_y} &= f_{n_y}(\mathbf{p}; y, t, c)\ , \quad y_{n_y}(t_I) = y_{n_y}^I(\mathbf{p}; c). \end{aligned} \tag{2.26}$$

The initial values of the system of differential equations $y_1^I(\mathbf{p};c),\ldots,y_{n_y}^I(\mathbf{p};c)$ may depend on one or more of the system parameters to be estimated and on the concentration parameter c.

We then get the resulting least-squares problem

$$\begin{aligned}
&\min_{\mathbf{p}} \sum_{k=1}^{m_{exp}}\sum_{i=1}^{m_t}\sum_{j=1}^{m_c}(w_{ij}^k(h_k(\mathbf{p};\mathbf{y}(\mathbf{p};t_i,c_j),t_i,c_j)-\hat{h}_{ij}^k))^2,\\
&\mathbf{g}_{\geq}(\mathbf{p};\mathbf{y}(\mathbf{p};t_i,c_j))\geq\mathbf{0},\\
&\mathbf{p}_L\leq\mathbf{p}\leq\mathbf{p}_U,\ \mathbf{p}\in\mathbb{R}^{n_u},
\end{aligned}\tag{2.27}$$

where $\mathbf{y}(\mathbf{p};t,c)$ is a solution vector of the system of n_y ODEs (2.26).

2.2.3 DAEs

We begin with m_{exp} data sets,

$$(t_i,c_j,\hat{h}_{ij}^k),\ i=1,\ldots,m_t,\ j=1,\ldots,m_c,\ k=1,\ldots,m_{exp},\tag{2.28}$$

where m_t time values, m_c concentration values, and $M=m_tm_cm_{exp}$ corresponding measurement values are given. We now add algebraic equations to the system of differential equations (2.26). The resulting fitting criterion $\mathbf{h}(\mathbf{p};\mathbf{y}(\mathbf{p};t,c),\mathbf{z}(\mathbf{p};t,c),t,c)$ depends on n_y differential variables $\mathbf{y}(\mathbf{p};t,c)$ and n_z algebraic variables $\mathbf{z}(\mathbf{p};t,c)$. The system of equations is now

$$\begin{aligned}
\dot{y}_1 &= f_1(\mathbf{p};\mathbf{y},\mathbf{z},t,c)\ , \quad y_1(t_I)=y_1^I(\mathbf{p};c),\\
&\vdots\\
\dot{y}_{n_y} &= f_{n_y}(\mathbf{p};\mathbf{y},\mathbf{z},t,c)\ , \quad y_{n_y}(t_I)=y_{n_y}^I(\mathbf{p};c),\\
0 &= g_1(\mathbf{p};\mathbf{y},\mathbf{z},t,c)\ , \quad z_1(t_I)=z_1^I(\mathbf{p};c),\\
&\vdots\\
0 &= g_{n_z}(\mathbf{p};\mathbf{y},\mathbf{z},t,c)\ , \quad z_{n_z}(t_I)=z_{n_z}^I(\mathbf{p};c).
\end{aligned}\tag{2.29}$$

The initial values of the differential equations $y_1^I(\mathbf{p};c),\ldots,y_{n_y}^I(\mathbf{p};c)$ and algebraic equations $z_1^I(\mathbf{p};c),\ldots,z_{n_z}^I(\mathbf{p};c)$ may depend on the system parameters to be estimated and on the concentration parameter c.

The inequality constraints take the form

$$\mathbf{g}_{\geq}(\mathbf{p};\mathbf{y}(\mathbf{p};t,c),\mathbf{z}(\mathbf{p};t,c))\geq\mathbf{0},\tag{2.30}$$

where $\mathbf{y}(\mathbf{p};t,c)$ and $\mathbf{z}(\mathbf{p};t,c)$ represent the solution of the system of DAEs (2.29), depending on the parameter vector $\mathbf{p}$ to be estimated and certain time and concentration values t and c, respectively.

The resulting least-squares problem for DAEs is

$$\begin{aligned} &\min_{\mathbf{p}} \sum_{k=1}^{m_{exp}} \sum_{i=1}^{m_t} \sum_{j=1}^{m_c} (w_{ij}^k (h_k(\mathbf{p};\mathbf{y}(\mathbf{p};t_i,c_j),\mathbf{z}(\mathbf{p};t_i,c_j),t_i,c_j) - \hat{h}_{ij}^k))^2, \\ &\mathbf{g}_{\geq}(\mathbf{p};\mathbf{y}(t_i,c_j),\mathbf{z}(\mathbf{p};t_i,c_j)) \geq \mathbf{0}, \\ &\mathbf{p}_L \leq \mathbf{p} \leq \mathbf{p}_U,\ \mathbf{p} \in \mathbb{R}^{n_u}, \end{aligned} \tag{2.31}$$

where $\mathbf{y}(\mathbf{p};t,c)$ and $\mathbf{z}(\mathbf{p};t,c)$ are solution vectors of the system of $n_y + n_z$ DAEs (2.29). The system is called an index-1 problem or an index-1 DAE if the algebraic equations can be solved (at least in principle) for $\mathbf{z}$ for all t. This is possible if and only if the Jacobian matrix

$$\mathbf{g}_z(\mathbf{p};\mathbf{y}(\mathbf{p};t,c),\mathbf{z}(\mathbf{p};t,c),t,c) \tag{2.32}$$

has full rank. In general, one definition of the *differential index* of a system of DAEs is the minimum number of differentiations with respect to t required to reduce the system to one consisting only of ODEs [2].

In EASY-FIT, there are implicit solvers that are able to solve index-2 DAEs and index-3 DAEs by transforming these high-index problems to index 1 by successive differentiation of the algebraic equations [26]. However, when using simple constraint differentiation to reduce the index of a DAE, the *drift* in the constraint (the amount by which it fails to be satisfied) increases linearly in time when reducing the index by 1 and quadratically in time when reducing the index by 2. If not done in a stable fashion, a numerical solution based on an index reduction from 3 to 0 may completely destroy the algebraic constraint. The transformation is performed because of the generic numerical instability of high-index DAEs. Although a straightforward application of index reduction is not stable, there exist stable numerical methods to solve high-index DAEs; see e.g., [21].

One method used to stabilize index reduction is called the *projected descriptor method* [1]. This method is complex; see [47]. It is also possible to solve high-index DAEs by applying a special discretization technique [25] involving an implicit method based on collocation; e.g., RadauIIA, a fully implicit, stiffly accurate Runge–Kutta method with 3 stages and order 5 [26].

Collocation methods are methods for which the solution of the system (2.26) is approximated by a polynomial that satisfies the differential equations at s distinct points. Runge–Kutta methods based on collocation at Radau points are collocation methods with order $2s-1$ and correspond to quadrature rules where one end of the interval is included. In the RadauIIA method, the right end of the interval is included as a collocation point.

2.2.4 Breakpoints

In practice it is common for the dynamics of a system to change during the trajectory. Accordingly, conditions on the solution at these *breakpoints* must be enforced, e.g., continuity of the solution. A typical example is a pharmacokinetic application with an initial infusion of a drug followed by subsequent infusions of drug doses by injection. Each infusion of a drug dose represents a breakpoint in the model because the dynamics depend on the amount of drug dose injected at each infusion. As is the case in this example, it is possible that the solution (i.e., the concentration of drug in the blood stream) is discontinuous at a breakpoint. *Variable breakpoints* are breakpoints that are treated as parameters to be optimized.

For simplicity we assume an ODE model with initial values as given in (2.26). Breakpoints are defined similarly for the DAE model (2.29). Formally we describe the model by

$$\begin{aligned} \dot{y}_1^{(1)} &= f_1^{(1)}(\mathbf{p}; y^{(1)}, t, c)\ , \quad y_1^{(1)}(t_I) = \bar{y}_1^{(1)}(\mathbf{p}; c), \\ &\vdots \\ \dot{y}_{n_y}^{(1)} &= f_{n_y}^{(1)}(\mathbf{p}; y^{(1)}, t, c)\ , \quad y_{n_y}^{(1)}(t_I) = \bar{y}_{n_y}^{(1)}(\mathbf{p}; c), \end{aligned} \tag{2.33}$$

for $t_I \le t \le \tau_1$, and

$$\begin{aligned} \dot{y}_1^{(K)} &= f_1^{(K)}(\mathbf{p}; y^{(K)}, t, c)\ , \quad y_1^{(K)}(\tau_K) = \bar{y}_1^{(K)}(\mathbf{p}; c, y_1^{(K-1)}(\mathbf{p}; \tau_K, c)), \\ &\vdots \\ \dot{y}_{n_y}^{(K)} &= f_{n_y}^{(K)}(\mathbf{p}; y^{(K)}, t, c)\ , \quad y_{n_y}^{(K)}(\tau_K) = \bar{y}_{n_y}^{(K)}(\mathbf{p}; c, y_{n_y}^{(K-1)}(\mathbf{p}; \tau_K, c)), \end{aligned} \tag{2.34}$$

for $\tau_{K-1} \le t \le \tau_K,\ K = 2, \ldots, N = m_b + 1$, where m_b is the number of breakpoints τ_{K-1}, and $\tau_0 = t_I < \tau_1 < \cdots < \tau_{m_b} < t_F$. The initial values of each subsystem are given by

$\bar{y}^{(K)}(\mathbf{p}; c, y^{(K-1)}(\mathbf{p}; \tau_K, c))$ and may depend on the parameter vector $\mathbf{p}$ to be estimated, the concentration value c, and the solution of the previous phase with breakpoint τ_{K-1}. In the numerical algorithm for the solution to such problems, the integration of the differential equations must be restarted at each breakpoint.

2.3 A Translator for Optimal Control Problems

In computer science and linguistics, *syntax analysis* is the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given formal grammar. In computer science, a *source-to-source translator* is a type of compiler that takes code in a high–level language as its input and outputs code in another high–level language.

In this thesis, we are concerned with a simple version of a translator that does not form a component of a compiler. Recall, EF2$\mathbb{SOCS}$ provides access to all the problem descriptions and solutions of the parameter estimation problems in the database of EASY-FIT, allows us to specify new problems in the modelling language PCOMP, and avoids most if not all manual coding of FORTRAN programs to run with $\mathbb{SOCS}$. In the present case, the syntax analysis involves analyzing PCOMP code and translating it into suitable $\mathbb{SOCS}$ code. However, because the PCOMP code is essentially FORTRAN77 syntax, often no translation is required to yield the desired input for $\mathbb{SOCS}$. Because FORTRAN is backward compatible, FORTRAN77 code is supported by the FORTRAN90 language used for $\mathbb{SOCS}$. Only the PCOMP declarations discussed in Section 3.1 require special attention. Each PCOMP declaration is treated by a specialized function in the translator. These functions are designed to recognize only the exact declaration format as described in Section 3.1. They contain the set of rules that correctly translate the PCOMP declaration into the appropriate format for $\mathbb{SOCS}$. All other PCOMP code remains unchanged during the translation process.

Chapter 3

Translator Functionality

The $\mathbb{SOCS}$ software package remains relatively untested in terms of solving a large test set of parameter estimation problems and evaluating the results. Although parameter estimation problems are common in industry and may easily be found in the literature, there is also a single source that supplies a large test set of problems of the explicit model, ODE, and DAE varieties. This source is the database of problems included in the parameter estimation software EASY-FIT. As part of the testing process of $\mathbb{SOCS}$, we ultimately wish to compare the results of $\mathbb{SOCS}$ to EASY-FIT. This will provide us with a benchmark with which to judge how well $\mathbb{SOCS}$ performs in comparison to EASY-FIT.

EASY-FIT presently has 826 parameter estimation problems of the sorts described in Chapter 2. These problems are coded in a high-level language called PCOMP. Because PCOMP allows for much shorter code in specifying parameter estimation problems and is similar in syntax to FORTRAN77, we would like to make use of this language to specify parameter estimation problems in $\mathbb{SOCS}$. To make use of this language, we design a translator, called EF2$\mathbb{SOCS}$, to translate these problems into the input required for $\mathbb{SOCS}$. The user may wish to investigate using different strategies to solve parameter estimation problems, such as using different initial guess methods for the state variables. This may be quickly and easily accomplished through the GUI in EF2$\mathbb{SOCS}$. This provides EF2$\mathbb{SOCS}$ with elements of a *problem-solving environment* (PSE).

According to Rice and Boisvert, a PSE is a computer system that provides all the computational facilities necessary to solve a target class of problems efficiently [42]. The facilities include advanced solution methods, automatic selection of solution methods, and ways to easily incorporate novel solution methods. PSEs solve simple or complex problems, support both rapid prototyping

and detailed analysis, and they can be used both in pedagogy or at the frontiers of scientific research. PSEs play an important role in today's scientific research. Some examples of PSEs include MATLAB, Maple, and Mathematica.

The functionality of EF2$\mathbb{SOCS}$ is essential when considering the amount of time required to code even the simplest of parameter estimation problems in $\mathbb{SOCS}$. A typical $\mathbb{SOCS}$ code used to solve a simple parameter estimation problem can easily reach 500 lines of FORTRAN77 code. It is therefore unrealistic to code each problem manually in $\mathbb{SOCS}$. This automated process of reading input from EASY-FIT and writing an output file that is directly executable by $\mathbb{SOCS}$ saves time as well as a significant number of potential coding errors on the part of the user when it comes to solving such a large number of problems with $\mathbb{SOCS}$.

In this chapter, we consider the input requirements for EASY-FIT and $\mathbb{SOCS}$. We give a complete list of the function, variable, and other declarations used in the EASY-FIT input that are supported EF2$\mathbb{SOCS}$. We talk about the EF2$\mathbb{SOCS}$ functionality, including its GUI. Finally, we present a sample EASY-FIT input file for an explicit model function along with the output file generated by EF2$\mathbb{SOCS}$.

3.1 Input Format for EASY-FIT

This section gives a list of all declarations supported by the modelling language used by EASY-FIT, called PCOMP [45]. For a more general and complete description of PCOMP, see [45], [14]. All model functions are defined in the PCOMP modelling language, and they are read in, translated by EASY-FIT, and compiled by EASY-FIT into FORTRAN77 code behind the scenes. The PCOMP-language is a subset of FORTRAN77 with a few extensions. In particular, the declaration and executable statements must satisfy the usual FORTRAN77 input format; e.g., they must start at column 7 or later. A statement line is read in until column 72. Comments, denoted with a `C` in the first column, may be included in a program text wherever needed. Continuation marks, denoted with a `/` in column 6, must be used to continue statements on subsequent lines. Either capital or small letters are interchangeable in identifiers of the user and key words of the language; i.e.,

PCOMP is not case sensitive. Note that each line of PCOMP code is automatically converted to capital letters by EF2𝕊𝕆ℂ𝕊. The length of an identifier must be no more than 20 tokens.

In PCOMP, most variables are declared implicitly by their assignment statements. PCOMP possesses special constructs to identify program blocks, denoted with a * in the first column, e.g., `* PARAMETER`. The only constructs permitted in PCOMP are listed and described below.

* `PARAMETER`

  This declaration specifies constant integer parameters to be used throughout the program, particularly for dimensioning index sets; see the index set description following PCOMP constructs.

* `SET OF INDICES`

  This declaration defines index sets that can be used to declare data, variables, and functions or to define `SUM` or `PROD` statements, e.g.,

  ```
  ind = 1..10
  ```

* `INDEX`

  This declaration defines an index variable that can be used in a `FUNCTION` program block.

* `REAL CONSTANT`

  This declaration defines real data, either without index or with a one- or two-dimensional index. An index may be a variable or a constant number within an index set. Arithmetic expressions may also be included, e.g.,

  ```
  b(i)=0.1*i, i in ind
  a(i,j)=1/(i+j-1), i in ind, j in ind
  ```

* `INTEGER CONSTANT`

  This declaration defines integer data, either without an index or with a one- or two-dimensional index. An index may be a variable or a constant number within an index set. Arithmetic integer expressions may also be included, e.g.,

  ```
  b(i)=2*i, i in ind
  ```

```
a(i,j)=2*(i+j), i in ind, j in ind
```

* `TABLE <identifier>`

  This declaration defines a one- or two-dimensional array of constant real numbers. In subsequent lines, one has to specify one or two indices followed by one real value per line in a free format (starting at column 7 or later).

* `VARIABLE`

  This declaration defines a real variable with up to one index.

* `CONINT <identifier>`

  This declaration defines a piecewise constant interpolation function.

* `LININT <identifier>`

  This declaration defines a piecewise linear interpolation function.

* `SPLINE <identifier>`

  This declaration defines a spline interpolation function.

* `MACRO <identifier>`

  This declaration defines a macro function, i.e., an arbitrary set of PCOMP statements that define an auxiliary function to be expanded inline in function declaration blocks. Macros are identified by a name that can be used in any right-hand side of an assignment statement.

* `FUNCTION <identifier>`

  This declaration defines a function. Functions must have at most one argument that must be an index variable for which function and derivative values are to be evaluated. The subsequent statements must assign a numerical value to the function identifier.

* `END`

  This declaration signals the end of the program.

If EF2$\mathbb{SOCS}$ finds a declaration that it does not recognize, i.e., one that is not found in the above list or one that does not have the appropriate format, a warning is given, and this program

block is ignored, likely resulting in an output file that is missing crucial information. An error is written to the EF2$\mathbb{SOCS}$ output file that prevents the $\mathbb{SOCS}$ code from compiling.

It is recommended, although not necessary, to follow the order of the above program blocks to avoid using a variable that has yet to be defined. All lines after the final `END` statement are ignored by EF2$\mathbb{SOCS}$. The statements within the program blocks are similar to FORTRAN77 notation; for more details, see [45]. We now give more precise descriptions for some of the above PCOMP declarations:

Index sets: Index sets are required for the `SUM` and `PROD` expressions, as described below, and for defining indexed data, variables, and functions. They can only be defined in the following 4 ways:

1. Range of indices, e.g.,

   ```
   ind1 = 1..27
   ```

2. Set of indices, e.g.,

   ```
   ind2 = 3,1,17,27,20
   ```

3. Computed index sets, e.g.,

   ```
   ind3 = 5*i + 100, i=1..n
   ```

4. Parameterized index sets defining a set of integers ranging from the previously declared endpoints `n` to `m`, e.g.,

   ```
   ind4 = n..m
   ```

Interpolation functions: In PCOMP, interpolation of user-defined data is used to create piecewise constant interpolants, piecewise linear interpolants, or cubic spline interpolants. We are given M pairs of real values $(t_1, y_1), \ldots, (t_M, y_M)$. In the case of a spline, we define a cubic piecewise interpolant.

Following [45], in the first case, we define a piecewise constant interpolant by:

$$c_{int}(t) = \begin{cases} 0, & t < t_1 \ , \\ y_i, & t_i \leq t < t_{i+1} \ , \ i = 1, \ldots, M \ , \\ y_N, & t_M \leq t \ . \end{cases}$$

A continuous piecewise linear interpolation function is defined by:

$$l(t) = \begin{cases} y_0, & t < t_1 \ , \\ y_i + \frac{t - t_i}{t_{i+1} - t_i}(y_{i+1} - y_i), & t_i \leq t < t_{i+1} \ , \ i = 1, \ldots, M-1 \ , \\ y_M, & t_M \leq t \ . \end{cases}$$

The choice of cubic spline used by PCOMP is somewhat unusual [46]. Accordingly, it is necessary that we first give some background on cubic splines.

In cubic spline interpolation, cubic polynomial pieces are combined to produce an overall interpolant. Mathematically, let $f(t)$ be a function on the interval $[t_I, t_F]$ and let

$$\Pi = \{t_I = t_1 < t_2 < t_3 < \cdots < t_{M-1} < t_M = t_F\}$$

be M equally spaced points at which $f(t)$ is to be interpolated; i.e., the t_i divide $[t_I, t_F]$ into $M - 1$ uniform subintervals, called a *partition* Π of $[t_I, t_F]$.

A cubic spline interpolant of $f(t)$ relative to the partition Π is a function $s(t)$ such that

1. on each subinterval $[t_j, t_{j+1}]$, $j = 1, 2, 3, \ldots, M-1$, $s(t)$ is the cubic polynomial

$$s_j(t) = A_j + B_j(t - t_j) + C_j(t - t_j)^2 + D_j(t - t_j)^3;$$

2. $s(t_j) = f(t_j)$, $j = 1, 2, \ldots, M$;

3. $s'(t)$ is continuous on $[t_I, t_F]$;

4. $s''(t)$ is continuous on $[t_I, t_F]$.

The interpolant $s(t)$ is continuous on $[t_I, t_F]$. The above four conditions are not enough to completely determine the interpolating function $s(t)$. The function $s(t)$ is composed of

$M-1$ different cubic polynomials, each with four coefficients, so there are a total of $4(M-1)$ unknowns. Interpolation provides M equations. Continuity of the spline and its first two derivatives contribute an additional $3(M-2) = 3M-6$ equations (continuity applies at the interior points $t_2, t_3, t_4, \ldots, t_{M-1}$ only). The definition of the spline therefore provides $(M) + 3(M-2) = 4M-6$ equations. Accordingly, in order to completely determine the interpolating function, two more conditions must be specified.

The three most common additional conditions are:

1. *not-a-knot*, where it is required that $s'''(t)$ be continuous at $t = t_2$ and $t = t_{M-1}$; an equivalent interpretation is that one cubic polynomial is fit on the first two subintervals and one more is fit on the last two subintervals;
2. *clamped*, where $f'(t_I)$ and $f'(t_F)$ are known so that $s'(t_I) = f'(t_I)$ and $s'(t_F) = f'(t_F)$;
3. *natural* (or *free*), where $s''(t_I) = s''(t_F) = 0$.

Once the final two conditions have been specified, a system of $4(M-1)$ equations in $4(M-1)$ unknowns is formed. Equations for A_j, B_j, C_j, and D_j follow from the definition of $s(t)$ by using the interpolation condition, continuity of $s(t)$, continuity of $s'(t)$, and continuity of $s''(t)$ respectively. After some algebraic manipulation of these equations, one obtains an equation that forms the basis for a tridiagonal system of equations for determining the C_j. This system is solved for the coefficients $C_2, ..., C_{M-1}$. We use the boundary conditions to get equations for C_1 and C_M. Using the C_j we can determine the values of the remaining coefficients using the equations for A_j, B_j, and D_j [12].

In order to create a cubic spline, there must be at least 4 data points. A not-a-knot spline is used on the first 4 data points. All subsequent data points are interpolated using a clamped spline, assigning a slope of zero at the right end point and a slope equal to that of the not-a-knot spline on the first 4 points at the fourth point. So, the 4 additional conditions are:

$$s_1'''(t_2) = s_2'''(t_2) \text{ and } s_2'''(t_3) = s_3'''(t_3)$$

$$s_4'(t_4) = s_3'(t_4) \text{ and } s_{M-1}'(t_M) = 0 \ .$$

In order to preserve the $4(M-1)$ equations when evaluating $s(t)$ for $t > t_4$, we note that the condition $s_4'(t_4) = s_3'(t_4)$ is contained in the continuity of $s'(t)$ and does not contribute a new condition. Also, the condition $s_4''(t_4) \neq s_3''(t_4)$ is not enforced, eliminating one of the equations enforcing $s''(t)$ to be continuous. Therefore, the resulting number of equations is $4(M-1)$.

According to [45], a zero derivative is enforced at the right end point because interpolated data are often based on experiments that reach a steady state, i.e., a constant value.

As an example, to interpolate the nonlinear function $f(t)$ by a piecewise constant interpolation of the discrete values $f(t_i) = y_i$ from Table 3.1, we define a program block starting with the keyword `CONINT` followed by the name of the function. The numerical time and function values are given on subsequent lines, using any standard FORTRAN77 format starting at column 7:

```
*       CONINT F
        0.0  0.00
        1.0  4.91
        2.0  4.43
        3.0  3.57
        4.0  2.80
        5.0  2.19
        6.0  1.73
        7.0  1.39
        8.0  1.16
        9.0  1.04
        10.0 1.00
```

The interpolation functions are treated as intrinsic FORTRAN77 functions; i.e., they have to contain a variable or constant as a parameter. So, if we assume that `T` has previously been

declared as a PARAMETER, for example, a valid statement would be

```
*     FUNCTION OBJ
      OBJ = F(T)
```

where the function OBJ is the piecewise constant interpolation function F evaluated at the point T.

Macros: In contrast to FORTRAN77, PCOMP does not allow for the declaration of subroutines. Alternatively, one may define a macro, i.e., an arbitrary sequence of PCOMP statements that define a variable to be expanded inline in subsequent function declaration blocks. Macros are identified by a name that can be used in any right-hand side of an assignment statement

```
*     MACRO <identifier>
```

followed by a group of PCOMP statements that assign a numerical value to the identifier. Macros do not permit recursion and have no arguments, but they may access all previously declared variables, constants, or functions at time of declaration. New variables may be declared within a macro, and any values assigned to these variables are also available outside of the corresponding function block. If we assume that x is a variable, then to define a macro that computes the square of x, we would write

```
*     MACRO sqr
      sqr = x*x
```

We may now replace each occurrence of the term x*x with the macro sqr, for example

```
      f = sqr-5.2
```

SUM and PROD expressions: Sums and products over predetermined index sets are coded using SUM and PROD expressions, where the corresponding index and the index set must be specified. For example,

```
inda = 1..5

f = 100*SUM(p(i)**a(i), i IN inda)
```

where `p` could be an array containing the parameters to be estimated and defined by an index set and `a` is an array of constant data. In mathematical notation we are evaluating

$$f = 100 \sum_{i=1}^{5} p(i)^{a(i)} \ .$$

Note that EF2SOCS does not accept nested `SUM` or `PROD` expressions. Consequently, the user may need to expand these expressions manually.

Functions: Functions must be treated in a particular way when using index sets. Because the design goal of PCOMP is to generate short, efficient FORTRAN77 code, indexed function names can only be used specifically as described. In other words, if a set of functions is declared by

```
*     FUNCTION f(i), i IN index
```

then only an access to `f(i)` is permitted and not to `f(1)` or `f(j)`, for example. Therefore, a reference to only a single function in the array of functions `f(i)` is not possible. The array may only be accessed as a whole; i.e., PCOMP does not extend the indexed functions to a sequence of single expressions as is done with `SUM` and `PROD` statements.

The purpose of EF2SOCS is not only to translate input from EASY-FIT to SOCS but also to use PCOMP to specify problems for SOCS. Therefore, it is worth noting that in using PCOMP code for specifying a problem for SOCS, other than the required syntax for PCOMP declarations described above, any other code that compiles in FORTRAN90 can be handled by EF2SOCS. For example, if one wanted to use a different spline function than the one included in PCOMP, it is possible to call such a function from PCOMP. The code for this function would not be written in PCOMP, but rather in a separate FORTRAN90 file not translated by EF2SOCS.

The user does not need to be concerned with using variable names that are also used in the SOCS software code. EF2SOCS has a list of these key words and automatically converts any

i	t_i	y_i
1	0.0	0.00
2	1.0	4.91
3	2.0	4.43
4	3.0	3.57
5	4.0	2.80
6	5.0	2.19
7	6.0	1.73
8	7.0	1.39
9	8.0	1.16
10	9.0	1.04
11	10.0	1.00

Table 3.1: Interpolation data

PCOMP variable names that conflict with the key words. A global conversion, i.e., all occurrences in the PCOMP file, is performed by appending an "X" to the end of the PCOMP variable name. A pre-processor in the form of a perl script is used to accomplish this task.

The following 3 subsections deal with the input format of the model functions that must be defined in EASY-FIT using the PCOMP language. EF2𝕊𝕆ℂ𝕊 has been designed to follow the same rules as PCOMP; based on the importance of understanding these rules, we provide them as a reference. Again, for a more complete description of the rules for each of the following 3 models, see [45].

Many problem-specific constants such as the number of unknown parameters, the number of concentration variables, and the number of fitting functions, etc., are contained in a file separate from the PCOMP code. These constants define the number of the inputs to be read from the PCOMP code. We describe this file after giving the rules for the input format of the model functions.

3.1.1 Input of Explicit Model Functions

To define explicit model functions in PCOMP, certain guidelines for the declaration of parameters and functions must be followed. The order in which these items are defined is essential for the interface between the input file and the data-fitting code. For defining variables, we have the following rules:

- The first variable names are identifiers for the n_u independent parameters to be estimated, $p_1, \ldots, p_{n_u}$.
- If a concentration variable c exists, then a corresponding variable name must be added next.
- The last variable name identifies the independent (time) variable t for which measurements are available.
- No other variables may be declared.

Similarly, there are rules for the order in which model functions are defined:

- First, m_{exp} fitting criteria $h_1(\mathbf{p}; t, c), \ldots, h_{m_{exp}}(\mathbf{p}; t, c)$ must be defined depending on $\mathbf{p}$, t, and optionally on c.
- The subsequent n_g functions are the constraints $g_1(\mathbf{p}; t, c), \ldots, g_{n_g}(\mathbf{p}; t, c)$, if they exist. They may depend only on the parameter vector $\mathbf{p}$ to be estimated, and certain time and concentration values t and c, respectively.
- No other functions may be declared.

The constants n_u, m_{exp}, and n_g are defined in the database of EASY-FIT. These constants along with many other problem-specific constants are contained in a file separate from the PCOMP code. Each problem has associated with it a PCOMP file as well as a file defining these constants.

3.1.2 Input of ODEs

For defining variables, we have the following rules:

- The first variables are identifiers for the n_u independent parameters to be estimated, $p_1, \ldots, p_{n_u}$.
- The subsequent n_y names identify the state variables of the system of ODEs, $y_1, \ldots, y_{n_y}$.
- If a concentration variable c exists, then a corresponding variable must be added next.
- The last variable name identifies the independent (time) variable t, for which measurements are available.
- No other functions may be declared.

Similarly, we have rules for the order in which model functions are defined:

- The first n_y functions are the right-hand sides of the system of differential equations, $f_1(\mathbf{p}; \mathbf{y}, t, c), \ldots, f_{n_y}(\mathbf{p}; \mathbf{y}, t, c)$, respectively.
- The subsequent n_y functions define the initial values, which may depend on the parameters to be estimated and the concentration variable, $y_1^I(\mathbf{p}; c), \ldots, y_{n_y}^I(\mathbf{p}; c)$.
- Next, m_{exp} fitting functions $h_1(\mathbf{p}; \mathbf{y}, t, c), \ldots, h_{m_{exp}}(\mathbf{p}; \mathbf{y}, t, c)$ are defined depending on $\mathbf{p}$, $\mathbf{y}$, t, and c, where $\mathbf{y}$ denotes the state variable of the system of differential equations.
- The final $n_{\geq}$ functions are the inequality constraints $g_1(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c)), \ldots, g_{n_{\geq}}(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c))$, if they exist, where $\mathbf{y}(\mathbf{p}; t, c)$ represents the solution of the system of ODEs, depending on the parameter vector $\mathbf{p}$ to be estimated, and certain time and concentration values t and c, respectively.
- No other functions may be declared.

The constants n_u, n_y, m_{exp}, and $n_{\geq}$ are defined in the database of EASY-FIT. The last m_b of the n_u parameters to be estimated are understood to be breakpoints, if they have been declared. Also m_b, the number of constant or variable break points, are defined beforehand in the database of EASY-FIT.

3.1.3 Input of DAEs

The following order of PCOMP variables is required:

- The first variable names are identifiers for n_u parameters to be estimated, $p_1, \ldots, p_{n_u}$.
- The subsequent n_y names identify the differential variables $y_1, \ldots, y_{n_y}$.
- The subsequent n_z names identify the algebraic variables $z_1, \ldots, z_{n_z}$.
- If a concentration variable c exists, then a corresponding variable must be added next.
- The last variable name defines the independent (time) variable t for which measurements are available.
- No other functions may be declared.

Similarly, we have rules for the order in which the model functions are defined:

- The first n_y functions are the right-hand sides of the system of differential equations, $f_1(\mathbf{p}; \mathbf{y}, \mathbf{z}, t, c), \ldots, f_{n_y}(\mathbf{p}; \mathbf{y}, \mathbf{z}, t, c)$, respectively.
- The subsequent n_z functions are the right-hand sides of the algebraic equations, i.e., functions $g_1(\mathbf{p}, \mathbf{y}, \mathbf{z}, t, c), \ldots, g_{n_z}(\mathbf{p}, \mathbf{y}, \mathbf{z}, t, c)$.
- Subsequently, n_y functions define initial values for the differential equations, which may depend on the parameters to be estimated and the concentration variable, $y_1^I(\mathbf{p}; c), \ldots, y_{n_y}^I(\mathbf{p}; c)$.
- Then n_z functions define initial values for the algebraic equations, which may depend on the parameters to be estimated and the concentration variable, $z_1^I(\mathbf{p}; c), \ldots, z_{n_z}^I(\mathbf{p}; c)$.
- Next m_{exp} fitting functions $h_1(\mathbf{p}; \mathbf{y}, \mathbf{z}, t, c), \ldots, h_{m_{exp}}(\mathbf{p}; \mathbf{y}, \mathbf{z}, t, c)$ must be defined depending on $\mathbf{p}, \mathbf{y}, \mathbf{z}, t$, and c, where $\mathbf{y}$ and $\mathbf{z}$ are the differential and algebraic state variables of the DAE.
- The final $n_{\geq}$ functions are the constraints $g_1(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c), \mathbf{z}(\mathbf{p}; t, c)), \ldots, g_{n_{\geq}}(\mathbf{p}; \mathbf{y}(\mathbf{p}; t, c), \mathbf{z}(\mathbf{p}; t, c))$, if they exist, where $\mathbf{y}(\mathbf{p}; t, c)$ and

$\mathbf{z}(\mathbf{p}; t, c)$ represent the solution of the system of DAEs (2.29), depending on the parameter vector $\mathbf{p}$ to be estimated and certain time and concentration values t and c, respectively.

- No other functions may be declared.

The constants n_u, n_y, n_z, m_{exp}, and $n_{\geq}$ are defined in the database of EASY-FIT and must coincide with the corresponding numbers of variables and functions, respectively. The last m_b fitting variables are considered to be breakpoints if they have been declared beforehand in the database of EASY-FIT.

Note: Presently, EF2$\mathbb{SOCS}$ is not designed to accommodate high-index DAEs; i.e., DAEs with index greater than 1. EASY-FIT can handle high-index problems; see Section 2.2. However, $\mathbb{SOCS}$ cannot handle high-index DAEs. Because EF2$\mathbb{SOCS}$ cannot transform high-index DAEs to index-1 problems, these problems cannot be translated directly.

Many problem-specific constants are contained in a file for each problem. This files form part of the EASY-FIT database and are separate from the PCOMP code (see Appendix A). For the specific details of the content of the constants file, see [15]. We provide only a brief summary of the important parts of the file used by EF2$\mathbb{SOCS}$. If one exists, the capitalized identifier is given at the start of the line.

- model name and model type (1 for explicit models, 4 for ODEs, 5 for DAEs)
- NPAR: Number of unknown parameters (n_u), followed by number of variable breakpoints
- NRES: Number of constraints (equality and inequality: n_g)
- NEQU: Number of equality constraints ($n_{=}$)
- two doubles per line defining time value t and concentration value c to which the constraints (if present) are applied (time and concentration values for equality constraints are listed first, followed by time and concentration values for inequality constraints)
- NODE: Number of differential equations (n_y)
- NCONC: Number of concentration values (m_c)

- NTIME: Number of time measurements (m_t)
- NMEAS: Number of measurement sets (dimension of fitting function: m_{exp})
- METHOD: Order of ODE method, number of algebraic equations (n_z), number of index-1 DAEs, number of index-2 DAEs, number of index-3 DAEs
- parameter data (parameter names, lower bounds for parameters ($\mathbf{p}_L$), initial values for parameters ($\mathbf{p}_0$), upper bounds for parameters ($\mathbf{p}_U$))
- SCALE: Scale type for weight factors; i.e., global scaling strategy
- data (time values, concentration values (if any), observation values, weights: $(t_i, c_j, \hat{h}_{ij}^k, w_{ij}^k),\ i = 1, \ldots, m_t,\ j = 1, \ldots, m_c,\ k = 1, \ldots, m_{exp})$
- NDISCO: Number of constant breakpoints (m_b)
- list of breakpoint values, one per line ($\tau_i, i = 1, \ldots, m_b$)

Note that an option defining the type of norm used in the objective is omitted. All problems in the EASY-FIT database are solved using the L_2-norm, i.e., the square root of the sum of the squares of the residuals r_{ij}^k. Although it is possible to use alternative norms in EASY-FIT, because they cannot be specified in $\mathbb{SOCS}$, they are not included in EF2$\mathbb{SOCS}$.

3.2 Input Format for $\mathbb{SOCS}$

In this section we give a brief description of the subroutines used by $\mathbb{SOCS}$ to solve parameter estimation problems. For complete details of these and other subroutines in $\mathbb{SOCS}$, see [7]. The software for solving optimal control and parameter estimation problems can be divided into four parts:

1. the general optimal control routine HDSOPE, which is called by the user to solve parameter estimation problems, and the input routine HHSOCS;
2. the user-supplied subroutines needed to define the parameter estimation problem;

3. the optimization software needed to solve a sparse NLP subproblem;

4. the optimal control utility software available for special analysis and applications.

The following subroutines have generic names that may be changed by the user. In this description we use names that are consistent with those in the $\mathbb{SOCS}$ manual [7].

The user must define the problem using a routine called as ODEINP. This subroutine defines the phase-dependent problem input. It is called once for each phase. Recall, a phase is a portion of a trajectory in which the dynamics of the system remain unchanged. The dynamics of the system are described by a set of dynamic variables made up of the state variables and the control variables. For a multi-phase example, see Section 4.1. In this example, each value of the concentration variable defines a separate phase. Also defined in ODEINP is the discretization method used in the direct transcription algorithm. Given a discretization step size $\Delta t > 0$, many discretization methods are possible for this algorithm, with the second-order, i.e., $\mathcal{O}(\Delta t^2)$, trapezoidal method and fourth-order, i.e., $\mathcal{O}(\Delta t^4)$, Hermite–Simpson method being popular choices. The trapezoidal method is used as the default discretization method. For a description of these and other possible discretization methods, see [5]. Several options are also available for constructing an initial guess for the dynamic variables (2.1). Note that whenever initial or final values for the dynamic variables is missing, guesses for these values are provided by the $\mathbb{SOCS}$ code. For example, the guesses for the initial conditions of the control variables are set to 0; i.e, $\mathbf{u}(t_I) = 0$. Also, the guesses for the final conditions of the dynamic variables are set to the initial conditions; i.e, $\mathbf{d}(t_F) = \mathbf{d}(t_I)$. The initial guess method is defined in $\mathbb{SOCS}$ using the array INIT. Popular choices include:

INIT(1) = 1: Construct a linear initial guess between $\mathbf{d}(t_I)$ and $\mathbf{d}(t_F)$; i.e., linearly interpolate the boundary conditions. If $NGRID$ is the number of grid points used in the discretization, then for grid point j the independent variable is

$$t_j = (j-1)\left[\frac{t_F - t_I}{NGRID - 1}\right] + t_I, \tag{3.1}$$

and the dynamic variables are

$$\mathbf{d}(t_j) = (j-1)\left[\frac{\mathbf{d}(t_F) - \mathbf{d}(t_I)}{NGRID - 1}\right] + \mathbf{d}(t_I), \tag{3.2}$$

where $1 \leq j \leq NGRID$.

INIT(1) = 6: Construct an initial guess by solving an initial value problem (IVP) with a linear control approximation. The control variables $\mathbf{u}(t)$ are approximated by

$$\mathbf{u}(t) = \frac{t_F - t}{t_F - t_I}\mathbf{u}(t_I) + \frac{t - t_I}{t_F - t_I}\mathbf{u}(t_F). \tag{3.3}$$

The state variables $\mathbf{y}(t)$ are approximated by the solution of the ODE system

$$\dot{\mathbf{y}} = \mathbf{f}[\mathbf{y}(t), \mathbf{u}(t), t] \ , \tag{3.4}$$

solved as an IVP using the variable step size Runge–Kutta–Taylor integrator.

INIT(2) = 3: This option uses a variable order, variable step size backward differentiation formula (BDF). It is referred to as the stiff BDF integrator [20].

INIT(1) = 9: Construct an initial guess using linear interpolation/extrapolation of the M measurement data points (2.16) with grid points coinciding with the time values.

Subroutine ODERHS permits the user to define the right-hand sides of the DAEs, and nonlinear boundary conditions can be constructed in subroutine ODEPTF. Optional output can be constructed in subroutine ODEPRT. Parameter estimation problems require input of measurement data using the subroutine DDLOAD. ODEIGS is an optional user-supplied subroutine that defines the initial guess for the dynamic variables.

The following subroutines describe a collection of useful utility procedures available in the $\mathbb{SOCS}$ library that are commonly needed for many applications. In particular the subroutine AUXOUT is an auxiliary output utility that can be used to display the optimal control solution produced by $\mathbb{SOCS}$ at either a fixed step size during the phase or at a specified number of points. The subroutine OCSEVL is used to evaluate the optimal control solution at a few points. Subroutine AUXOUT may be more appropriate when the user wishes to display a complete time history of the solution. The primary function of OCSRNG is to construct estimates for the upper and lower bounds for the dynamic variables produced by $\mathbb{SOCS}$. This information is often useful when constructing scale information as well as for display purposes. The subroutine LINKST is useful

for linking dynamic variables across a phase boundary. Subroutines PHSLNG, PNTCON, and PTHCON are utility routines to simplify the specification of phase duration constraints, point functions, and path constraints, respectively. The organization of the subroutines used by $\mathbb{SOCS}$ to solve parameter estimation problems is illustrated in Figure 3.1. User-supplied subroutines are shown with double boxes. The optional subroutines are indicated by an asterisk. The user must call the $\mathbb{SOCS}$ algorithm HDSOPE and define the problem using the subroutine ODEINP. All other information is optional and may be supplied by the user or by using the dummy routines included in the $\mathbb{SOCS}$ library.

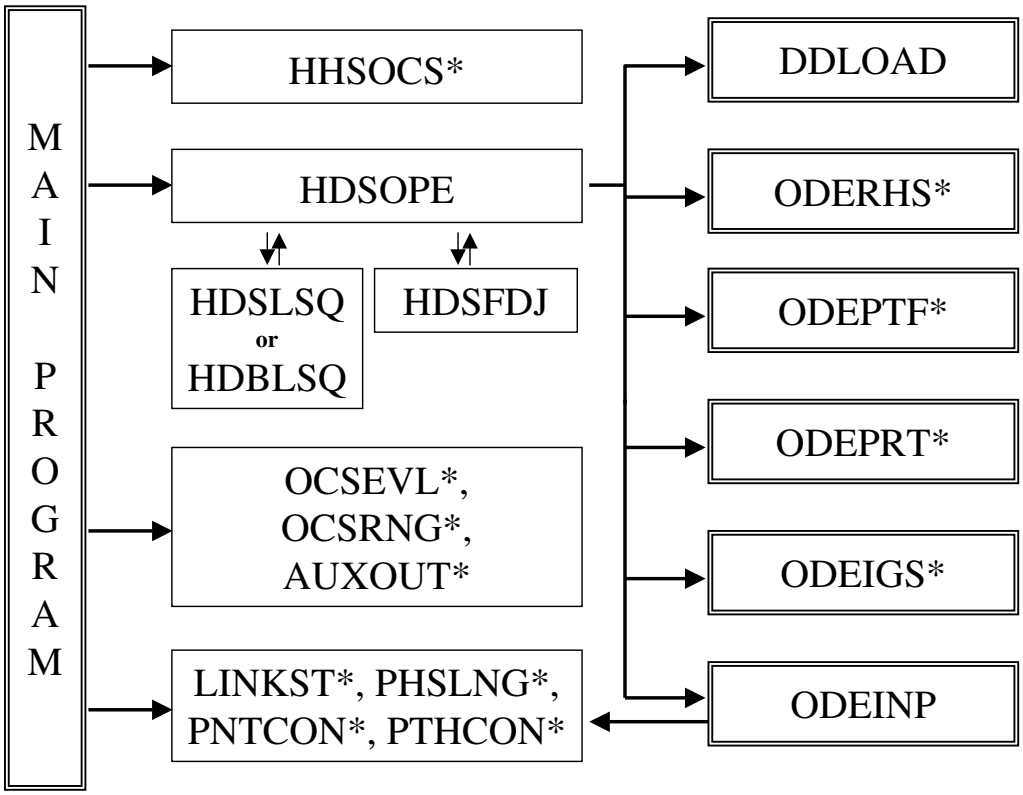


Figure 3.1: Organization of $\mathbb{SOCS}$ for Sparse Optimal Parameter Estimation

3.3 Sample Problem

We now have sufficient background to fully understand the translation of an EASY-FIT input file into a corresponding $\mathbb{SOCS}$ input file. As an example, we consider the simplest type of parameter estimation problem, an explicit model function with no constraint functions. For more examples, including parameter estimation problems involving differential equations, see Chapter 4.

The explicit model function we describe is called TP333 in the EASY-FIT software package. The experimental data can be found in Table 3.2. The two integers on the first line give the size of the measurement set. The first integer represents the number of rows in the measurement set, and the second integer represents the number of columns in the measurement set. The first column of data represents the time values, the second represents the observation values, and the third represents the weights associated with each measurement value. Other problem-specific constants are found in the data file in Appendix A. Note that the line numbers in this data file have been added for easy line identification. Following the problem formulation given in Section 2.1, we have $m_t = 8$ time values, $m_c = 1$ concentration value, $m_{exp} = 1$ measurement set, and $M = m_t m_c m_{exp} = 8$ corresponding measurement values.

We wish to fit parameters $\mathbf{p} = (p_1, p_2, p_3)^\mathsf{T}$ so that the data in Table 3.2 are approximated by the function

$$h(\mathbf{p}; t) = p_1 \exp(-p_2 t) + p_3 \ . \tag{3.5}$$

From Appendix A we see that the initial guess for the unknown parameters is $\mathbf{p}_0 = (30, 0.04, 3)^\mathsf{T}$ and the parameter bounds are $0 \le p_1 \le 1000$, $0 \le p_2 \le 1000$, and $0 \le p_3 \le 1000$. The least-squares data-fitting problem is

$$\begin{aligned} &\min_{\mathbf{p}} \ \sum_{i=1}^{M} (h(\mathbf{p}; t_i) - \hat{h}_i)^2, \\ &0 \le \mathbf{p} \le 1000, \ \mathbf{p} \in \mathbb{R}^3. \end{aligned} \tag{3.6}$$

The corresponding EASY-FIT PCOMP file is the following:

```
C-------------------------------------------------------
C
C     Problem:   TP333
C
C     Date:      02.03.1994
C
C-------------------------------------------------------
C
C   - Independent variables in the following order:
C     1. parameters to be estimated (p)
C     2. time variable (t)
C
*     VARIABLE
      p1, p2, p3, t
C
C-------------------------------------------------------
C
C   - Fitting criteria:
C
*     FUNCTION h
      h = p1*exp(-p2*t) + p3
C
C-------------------------------------------------------
C
*     END
C
C-------------------------------------------------------
```

8 3

4	72.1	1
5.75	65.6	1
7.5	55.9	1
24	17.1	1
32	9.8	1
48	4.5	1
72	1.3	1
96	0.6	1

Table 3.2: Measurement data for model TP333.

The EF2SOCS–generated input file for SOCS is given in Appendix B. EF2SOCS also creates a data file containing the measurement data. This data file is used by SOCS to solve TP333; see Table 3.2. Along with the call to HDSOPE, the parameter estimation solver in SOCS, the program also makes use of the subroutines EXPRHS, EXPINP, and EXPDDL.

In the subroutine EXPRHS, we define the model function. Because SOCS can only evaluate residuals on the state and/or algebraic variables and not functions of them, we need to introduce an algebraic variable into the equation defining the right-hand side of the fitting function. This equation then becomes a constraint that we define in the input subroutine. The algebraic variable is stored in the array `YVEC`, which is a real array containing the dynamic variables $\mathbf{y}(t)$ and $\mathbf{u}(t)$. EXPINP is used to define initial and final times, initial parameter values, parameter bounds, and the objective function. The user-defined subroutine INIEXP is called from EXPINP so that the data in Table 3.2 are loaded into the program. The data are then appropriately assigned to the correct variables in the subroutine EXPDDL. Here, the time values, measurement values, and weights are assigned. Because scaling of the residuals is needed, the subroutine EXPDDL also calculates the values for the weight array based upon the scaling option, which in this case is (2.21).

3.4 Software Architecture and Design

Software architecture deals with the design and implementation of the high-level structure of software. It is the result of assembling a number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system [33]. The major elements of a software architecture are *components* and *connectors*. The components are the building blocks of a software architecture, and the connectors describe how the components interact.

We make use use of two programming languages for EF2$\mathbb{SOCS}$. First, we choose the C programming language for the source-to-source translation because it is a portable language that is also one of the most commonly used languages for systems and applications programming. Second, we choose the Python programming language for the GUI because it is an interpreted language with a concise syntax, making the resulting programs easy to read and understand.

The primary organization of many software systems reflects the programming language in which the software is written. Because EF2$\mathbb{SOCS}$ is written in the C programming language, it easily adopts the hierarchical system using the main program with subroutines model of software architecture [19], as shown in Figure 3.2. The components are comprised of the procedures that have their own local data. The connectors are the procedures that have shared access to data declared in `main`. These data are only accessible to all procedures through procedure calls.

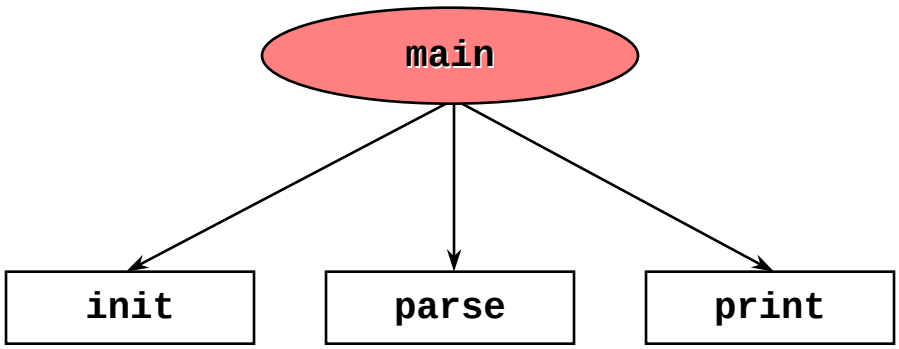


Figure 3.2: System overview of EF2$\mathbb{SOCS}$.

The `main` program is used to send the appropriate information, e.g., variables, to each of the procedures where the variables are to be defined. The first procedure called is `init`. Here, the problem-specific constants and measurement data are read in and assigned to the appropriate variables from `main`. Next, the PCOMP code for a given problem is translated using `parse`. Each declaration in the PCOMP code is read, and the appropriate rules, as given in Section 3.1, are applied. If EF2$\mathbb{SOCS}$ encounters any code that does not meet the requirements for PCOMP, then a warning is printed. Finally, the `print` procedure is used to emit the translated source. It is called with all the problem-specific information required to produce the $\mathbb{SOCS}$ code. This procedure contains dozens of print statements and checks many different conditions depending on the problem, e.g., the number of constraint functions, differential equations, concentration variables, etc.

Note that the GUI is not included in Figure 3.2. The GUI does not interact directly with EF2$\mathbb{SOCS}$ code; i.e., it does not make any direct calls to EF2$\mathbb{SOCS}$ C code. The GUI is a standalone application that runs the EF2$\mathbb{SOCS}$ executable file to translate PCOMP code into executable $\mathbb{SOCS}$ code. We now discuss the EF2$\mathbb{SOCS}$ GUI in greater detail.

3.4.1 The EF2$\mathbb{SOCS}$ GUI

The EF2$\mathbb{SOCS}$ GUI is written in the Python programming language and uses the toolkit wxPython. This toolkit enables Python programmers to create programs with a robust, highly functional GUI in a simple and easy manner. It is implemented as a Python extension module (native code) that wraps the popular wxWidgets cross-platform GUI library. Because wxPython is a cross-platform toolkit, EF2$\mathbb{SOCS}$ runs on multiple platforms with very few modifications; e.g., the sizes of some widgets are modified to create a consistent "look" for the GUI across platforms. Therefore, currently supported platforms for EF2$\mathbb{SOCS}$ are 32-bit Microsoft Windows, most Unix or Linux systems, and Macintosh OS X.

The structure of the EF2$\mathbb{SOCS}$ GUI in Figure 3.3 includes four major components:

1. The *Problem Editor* is a text window that defines the model equations using the PCOMP modelling language. Figure 3.4 shows the EF2$\mathbb{SOCS}$ Problem Editor. This editor provides

some text editing facilities, such as copy, paste, and line numbering to help the user to define their model quickly and conveniently. The user may create multiple instances of the Problem Editor for defining different problems simultaneously.

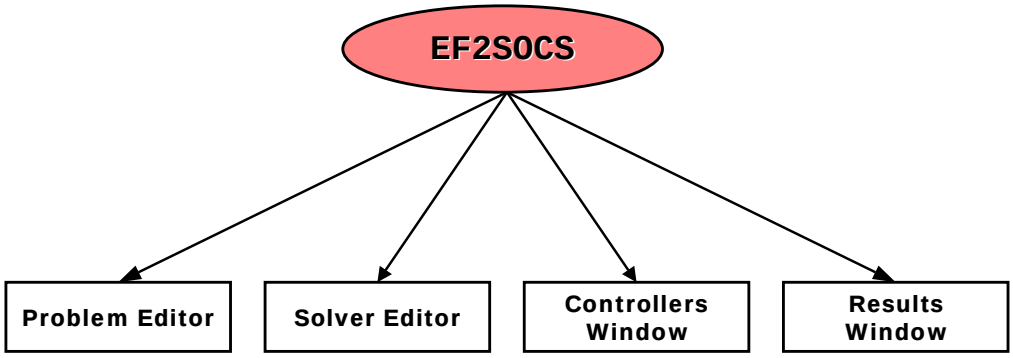


Figure 3.3: The overall structure of the EF2$\mathbb{S}\mathbb{O}\mathbb{C}\mathbb{S}$ GUI components.

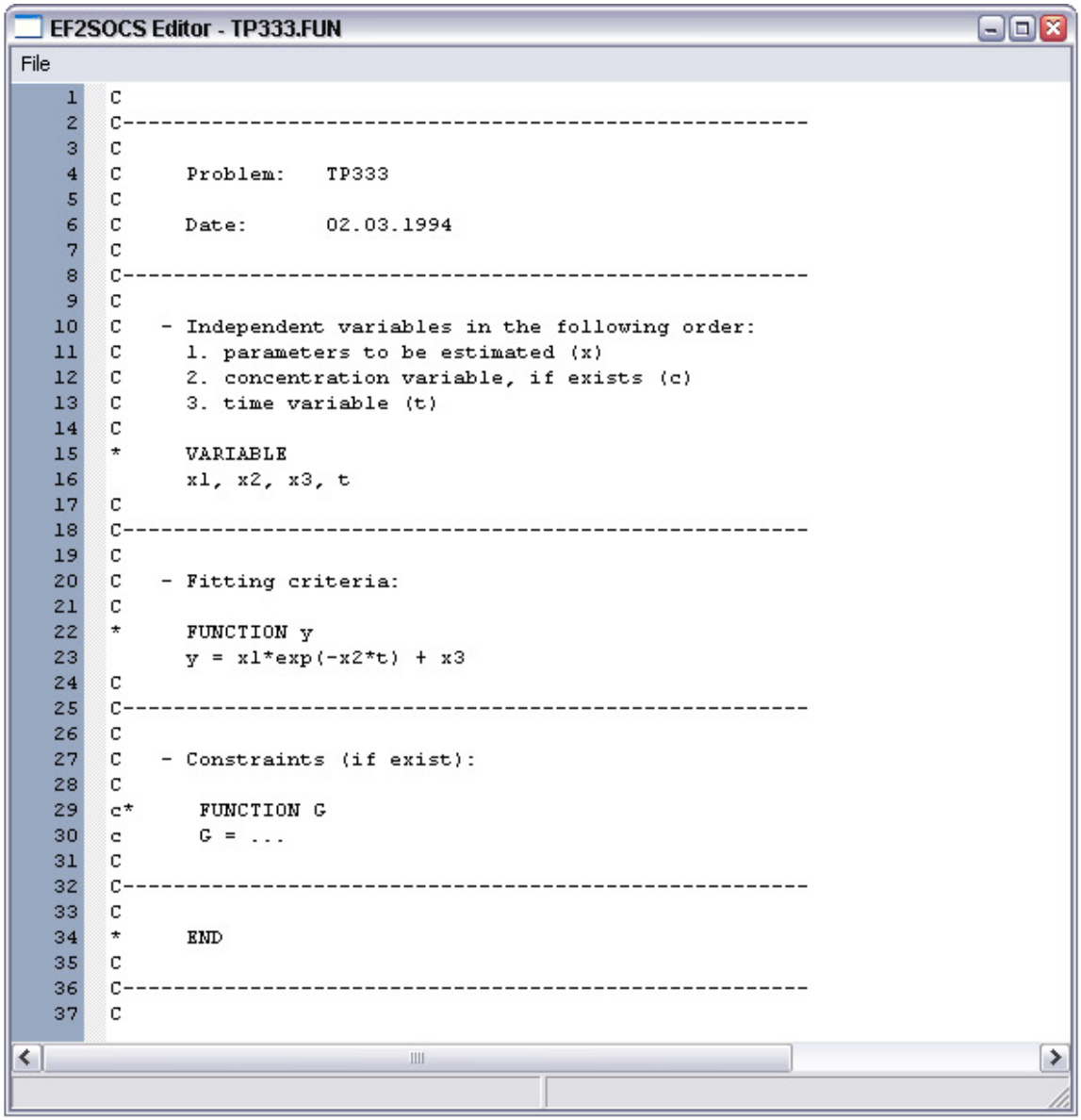
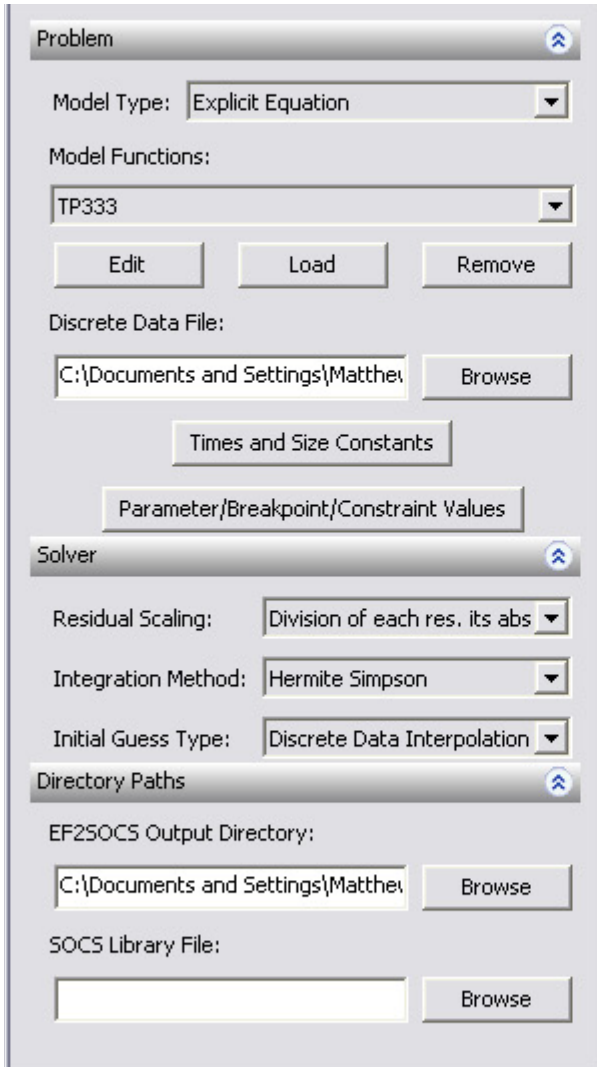


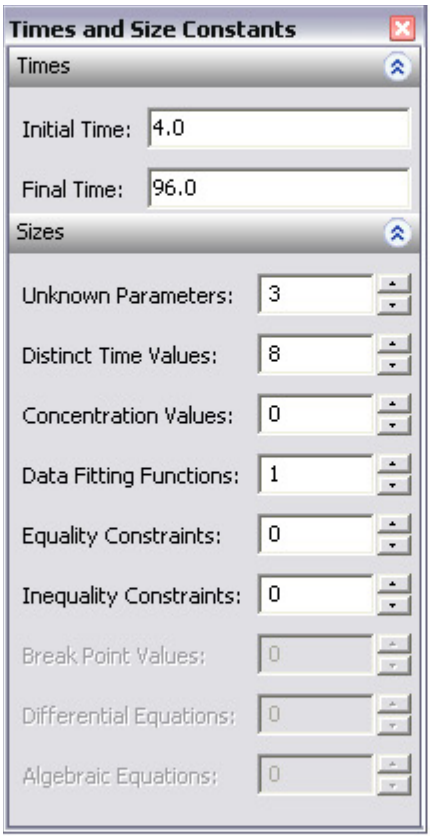
Figure 3.4: An instance of the EF2SOCS Problem Editor.

2. The *Solver Editor* defines various options used by SOCS in solving the parameter estimation problem. Figure 3.5 shows the EF2SOCS Solver Editor. This editor allows the user to specify an EF2SOCS controller that is composed of the residual scaling method using one of the options from Section 2.2.1. The available integration methods and initial guess types for the state variables are as described in the ODEINP SOCS subroutine in Section 3.2. The user may also specify the output directory for the translated PCOMP code and the SOCS output.



(a) Main Solver Editor.

Figure 3.5: An instance of the EF2SOCS Solver Editor.



(b) Sub-Solver Editor.

(c) Sub-Solver Editor.

Figure 3.5: An instance of the EF2𝕊𝕆𝕆𝕊 Solver Editor (cont.).

3. The *Controllers Window* shows a list of EF2𝕊𝕆ℂ𝕊 controllers specified by the user. Figure 3.6 shows the Controllers Window. The user may select an EF2𝕊𝕆ℂ𝕊 controller from the list to view its Problem Editor and Solver Editor settings, e.g., residual scaling method, the integration method, and the initial guess type for the state variables. The user may add or remove any number of EF2𝕊𝕆ℂ𝕊 controllers from the list. After defining and selecting a set of EF2𝕊𝕆ℂ𝕊 controllers, the user may either run the controllers to translate and solve the selected problems or run the controllers to only translate the selected problems. In both cases, the GUI creates a sub-process that runs the EF2𝕊𝕆ℂ𝕊 executable file to translate PCOMP code into executable 𝕊𝕆ℂ𝕊 code. If the user chooses to translate and solve the selected problems, the GUI will also create a sub-process to run the executable 𝕊𝕆ℂ𝕊 code with the 𝕊𝕆ℂ𝕊 library, producing 𝕊𝕆ℂ𝕊 output. The user may only want to produce the executable 𝕊𝕆ℂ𝕊 code by choosing "Translate" if, for example, they do not have access to the 𝕊𝕆ℂ𝕊 library on the current machine. A controller may also be stopped during the solving process. The Controllers Window shows the current status of each EF2𝕊𝕆ℂ𝕊 controller, such as the elapsed time of the running EF2𝕊𝕆ℂ𝕊 controller and whether the EF2𝕊𝕆ℂ𝕊 controller has terminated successfully. If a warning is output by EF2𝕊𝕆ℂ𝕊, it is printed at the beginning of the 𝕊𝕆ℂ𝕊 code and the controller will display a warning symbol.

4. The *Results Window* shows CPU time (in seconds) required of each EF2𝕊𝕆ℂ𝕊 controller on a given problem. Figure 3.7 shows an instance of the Results Window. Again, the user may select an EF2𝕊𝕆ℂ𝕊 controller from the list to view its settings. The user may save these results for further analysis by choosing "Save" from the File menu.

Therefore, the user may define the problem and load, save, and change solver settings easily through the GUI. Also, EF2𝕊𝕆ℂ𝕊 provides default settings so the user does not need to specifically define each solver setting. Table 3.3 shows the predefined settings in EF2𝕊𝕆ℂ𝕊.

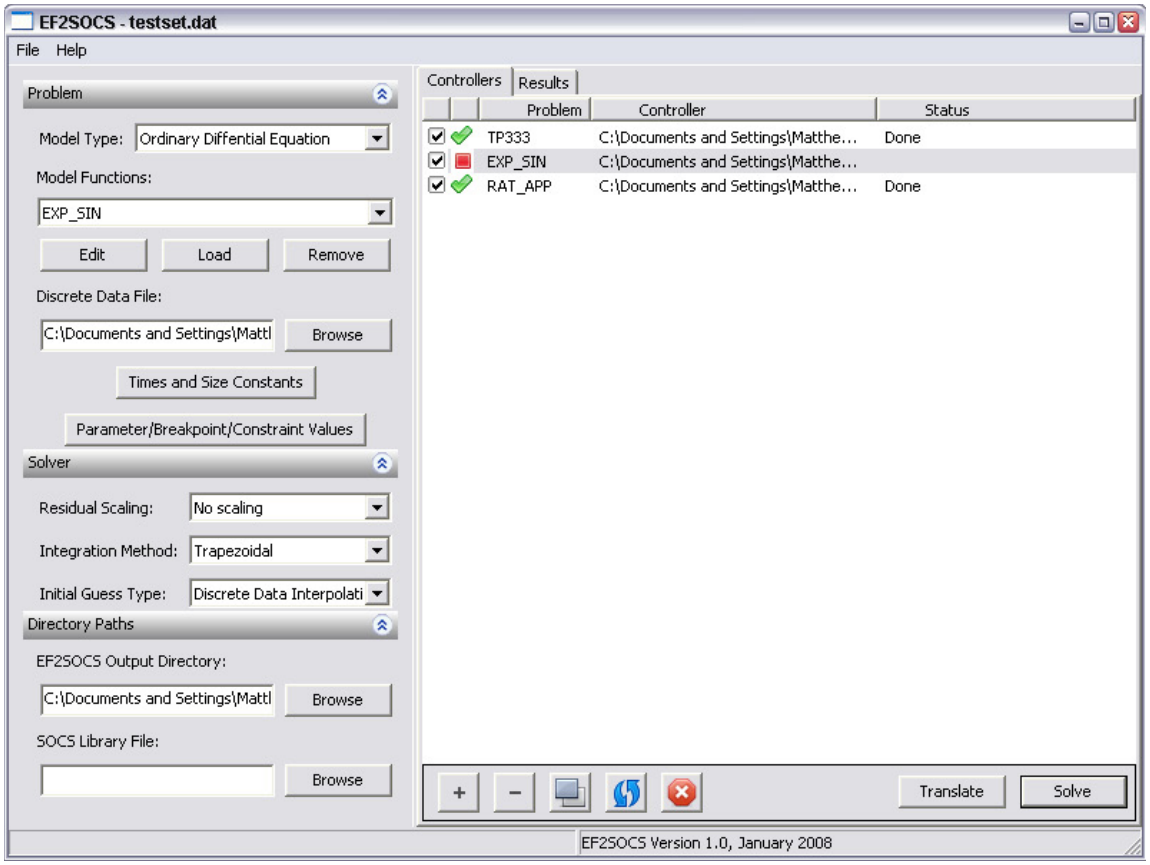


Figure 3.6: The EF2SOCS Controllers Window.

Settings	Options	Default Setting
Residual Scaling	no scaling, sum of squares, absolute value, squared value	no scaling
Integration Method	trapezoidal, Hermite–Simpson	trapezoidal
Initial Guess Type	linear, discrete data interpolation, IVP approximation, stiff BDF integrator	linear
EF2SOCS Output directory	any directory	current working directory

Table 3.3: Default settings for EF2SOCS GUI

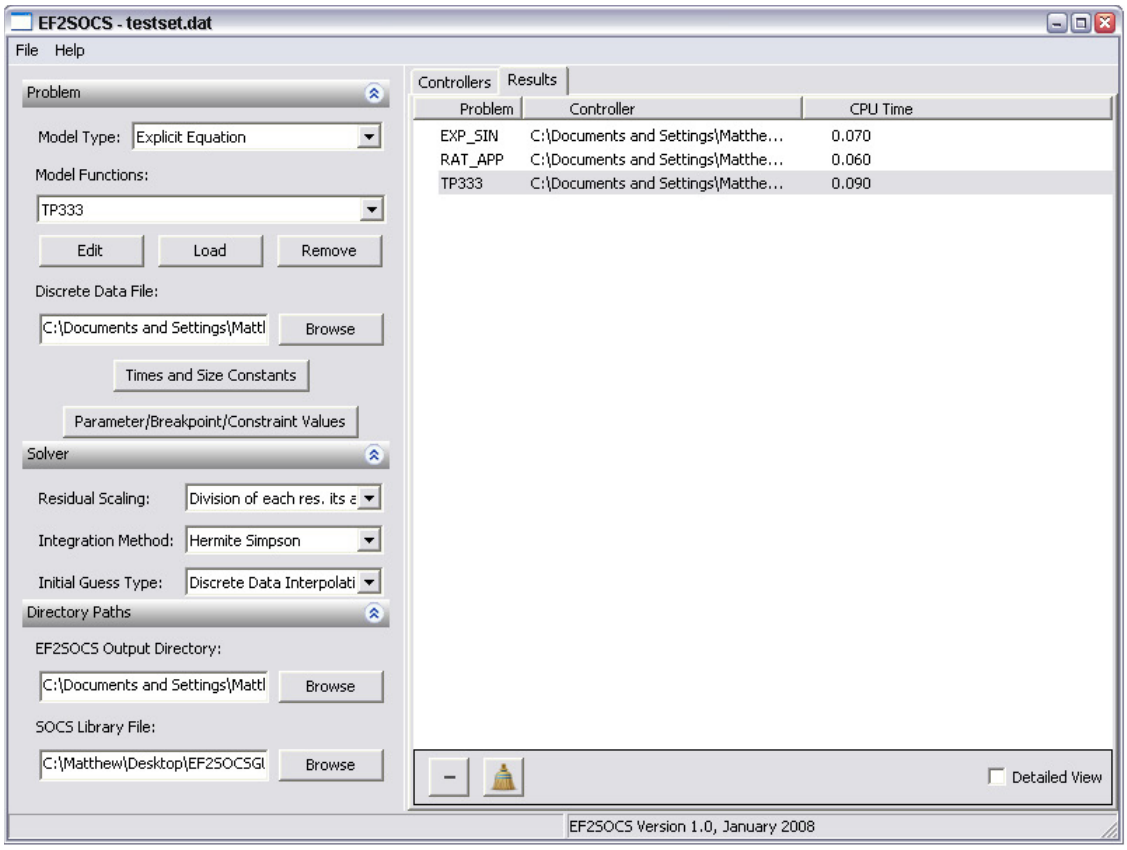


Figure 3.7: The EF2SOCS Results Window.

Chapter 4

Results and Discussion

To test EF2$\mathbb{SOCS}$ as well as the $\mathbb{SOCS}$ software package on a large number of problems, we run EF2$\mathbb{SOCS}$ on explicit model functions, ODEs, and DAEs. The EASY-FIT database contains 825 problems of these three types. However, not all of these problems can be translated by EF2$\mathbb{SOCS}$. We describe the subsets of the problems that cannot be translated and give the corresponding reasons. Excluding the problems that cannot be translated, we are left with a test set of size 691. A summary of the test results is given in the form of tables that we discuss in detail. First we describe in detail a single problem from each of the three categories of problems and present solution plots. We consider a $\mathbb{SOCS}$ *solution* to a problem to be the optimal set of unknown parameter values, state variable values, predicted model values, and objective function value as produced by $\mathbb{SOCS}$.

We begin with an example of an explicit model function called INTEG_X and the results obtained after using EF2$\mathbb{SOCS}$ to translate the EASY-FIT input file into $\mathbb{SOCS}$ input.

4.1 Sample Explicit Model Problem

In the EASY-FIT explicit model problem INTEG_X, we wish to fit simulated data. The model function (4.1) is evaluated at $M = m_t m_c m_{Exp} = 25$ points, and a uniformly distributed error of 5% is added to the function values using parameter values $\mathbf{p} = (1, 2, 3)^\mathsf{T}$. The unknown parameter vector is $\mathbf{p} = (p_1, p_2, p_3)^\mathsf{T}$. We wish to fit the unknown parameters and concentration variable c so that the data in Table 4.1 are approximated by the model function

$$h(\mathbf{p}; c, t) = \frac{p_1 \exp(-c)}{1 - p_2 \exp(-a) + p_3 \exp(-t)}. \tag{4.1}$$

The constant a is given the value 5.0. The initial guess for the unknown parameter vector is $\mathbf{p}_0 = (10, 10, 10)^\mathsf{T}$, and the parameter bounds are $0 \leq p_1 \leq 10$, $0 \leq p_2 \leq 20$, and $0 \leq p_3 \leq 20$. The least-squares data-fitting problem is

$$\begin{aligned} \min_{\mathbf{p}} \ & \sum_{i=1}^{m_t} \sum_{j=1}^{m_c} (w_{ij}(h(\mathbf{p}; c_j, t_i) - \hat{h}_{ij}))^2, \\ & 0 \leq p_1 \leq 10, \\ & 0 \leq p_2 \leq 20, \\ & 0 \leq p_3 \leq 20. \end{aligned} \tag{4.2}$$

This problem is solved using the default trapezoidal discretization method and the linear initial guess method for the dynamic variable given by (3.2). The solution is summarized in Table 4.2. The objective function values given by EASY-FIT and $\mathbb{SOCS}$ agree to 4 decimal places. Because of possible locally optimal solutions, the objective function values given by EASY-FIT and $\mathbb{SOCS}$ may agree to multiple decimal places despite a difference in parameter values. In an older version of EASY-FIT, different optimal solutions than those found by the current version may be given. For the problem INTEG_X, an alternative solution stored in the GUI of the previous version of EASY-FIT lists optimal parameter values for this problem to be $\mathbf{p}^* = (0.967, 8.372, 2.818)^\mathsf{T}$. It is worth noting that these optimal parameter values also give an objective function value that agrees to 4 decimal places with the one given by EASY-FIT and $\mathbb{SOCS}$.

A plot of the experimental data and $\mathbb{SOCS}$ solution to this explicit model problem is given in Figure 4.1. Due to the magnitude of the objective function value, the difference between the experimental data and theoretically predicted model function values is not noticeable. The colour depicts the magnitude of the numerical values of the fitting function $h(\mathbf{p}; c, t)$.

t_i	c_j	$\hat{h}_{ij}$	w_{ij}
1	1	0.183400496840477	1
2	1	0.26373416185379	1
3	1	0.338522046804428	1
4	1	0.353820204734802	1
5	1	0.368568271398544	1
1	2	0.061604518443346	1
2	2	9.24699977040291E-02	1
3	2	0.120332285761833	1
4	2	0.133212581276894	1
5	2	0.134147644042969	1
1	3	2.48860493302345E-02	1
2	3	0.034145575016737	1
3	3	4.56701554358006E-02	1
4	3	4.88160811364651E-02	1
5	3	4.85896691679955E-02	1
1	4	9.15580242872238E-03	1
2	4	1.29939131438732E-02	1
3	4	0.01550810970366	1
4	4	1.84268802404404E-02	1
5	4	1.84248797595501E-02	1
1	5	3.28793190419674E-03	1
2	5	4.76185046136379E-03	1
3	5	5.66617911681533E-03	1
4	5	6.43186643719673E-03	1
5	5	6.62047695368528E-03	1

Table 4.1: Experimental Data for problem INTEG_X.

Problem	**p** and obj. values	EASY-FIT	$\mathbb{SOCS}$
INTEG_X	p_1	0.970	0.972
	p_2	7.906	7.681
	p_3	2.828	2.833
	obj.	4.423×10^{-4}	4.242×10^{-4}

Table 4.2: Results obtained from EASY-FIT and $\mathbb{SOCS}$ for explicit model INTEG_X.

4.2 Sample ODE Problem

In the EASY-FIT ODE parameter estimation problem COMPET, we wish to fit simulated data that model a competition between two species, as described in [4]. The model functions (4.5) are evaluated at $M = m_t m_{Exp} = 50$ points, and a uniformly distributed error of 5% is added to the function values using parameter values $\mathbf{p} = (1, 1, 1, 0.99)^\mathsf{T}$. The unknown parameter vector is $\mathbf{p} = (p_1, p_2, p_3, p_4)^\mathsf{T}$. The model functions

$$h_1(\mathbf{p}; \mathbf{y}(\mathbf{p}; t), t)) \;=\; y_1, \tag{4.3}$$

$$h_2(\mathbf{p}; \mathbf{y}(\mathbf{p}; t), t)) \;=\; y_2, \tag{4.4}$$

depend on the solution $\mathbf{y}(\mathbf{p}; t)$ of a system of 2 ODEs

$$\begin{aligned} \dot{y}_1 &= p_3 \frac{y_1(1-y_1)}{2} - p_1 y_1 y_2, \quad y_1(0) = 0.02, \\ \dot{y}_2 &= p_4 \frac{y_2(1-y_2)}{2} - p_2 y_1 y_2, \quad y_2(0) = 0.02. \end{aligned} \tag{4.5}$$

The initial guess for the parameters is $\mathbf{p}_0 = (0.5, 0.5, 0.5, 0.5)^\mathsf{T}$ with lower and upper bounds given by $0 \le \mathbf{p} \le 100$. The least-squares data fitting problem is

$$\begin{aligned} &\min_{\mathbf{p}} \sum_{k=1}^{m_{Exp}} \sum_{i=1}^{m_t} (h_k(\mathbf{p}; \mathbf{y}(\mathbf{p}; t_i), t_i) - \hat{h}_i^k)^2, \\ &0 \le \mathbf{p} \le 100, \;\; \mathbf{p} \in \mathbb{R}^4. \end{aligned} \tag{4.6}$$

This problem is solved using the default trapezoidal discretization method and the linear initial guess method for the dynamic variables given by (3.2). The solution is summarized in Table 4.3.

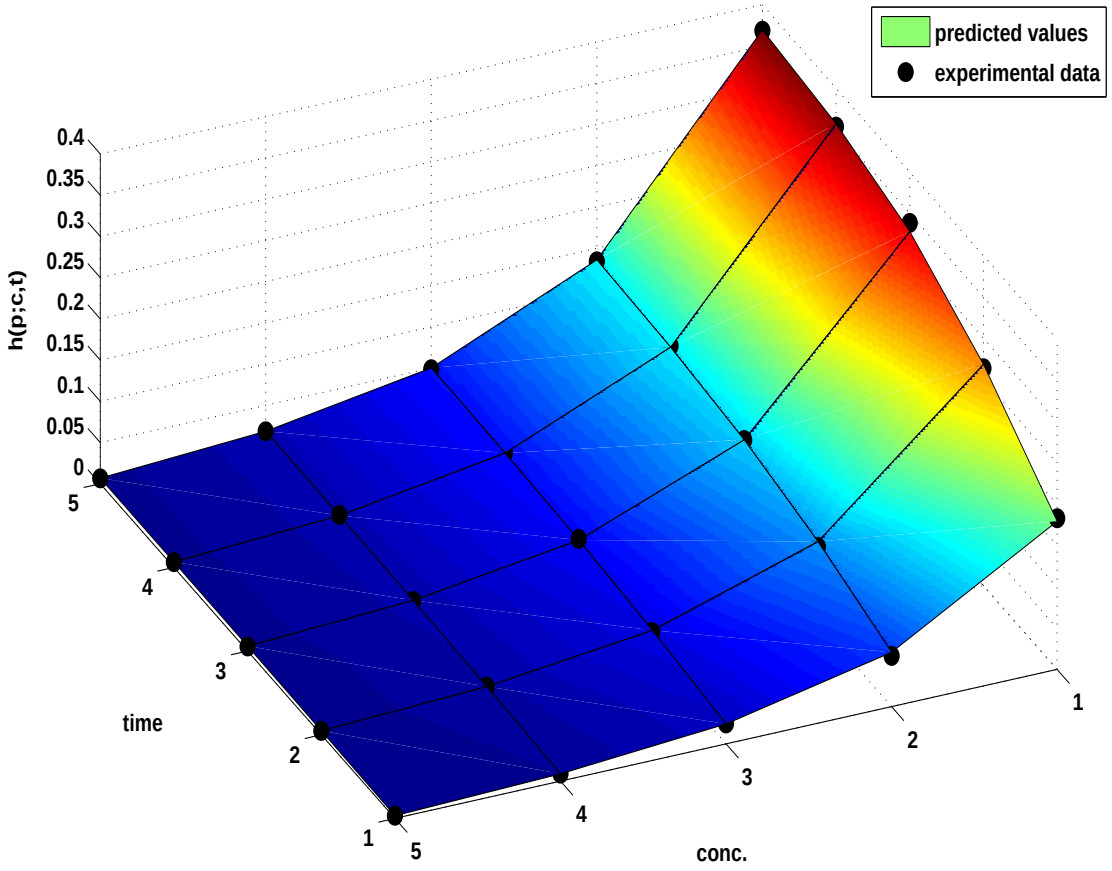


Figure 4.1: Plot of experimental data versus theoretically predicted model values obtained from $\mathbb{SOCS}$ for explicit model function INTEG_X.

All parameter values except p_1 agree to at least 3 decimal places. The objective function values found by EASY-FIT and $\mathbb{SOCS}$ also agree to 3 decimal places. All parameter values found by both software packages differ by less than 5% from those used to simulate the data. A plot of the experimental data and $\mathbb{SOCS}$ solution to this ODE parameter estimation problem is given in Figure 4.2.

Problem	**p** and obj. values	EASY-FIT	$\mathbb{SOCS}$
COMPET	p_1	0.985	0.983
	p_2	0.999	0.999
	p_3	0.996	0.996
	p_4	0.991	0.991
	obj.	0.0219	0.0218

Table 4.3: Results obtained from EASY-FIT and $\mathbb{SOCS}$ for ODE model COMPET.

4.3 Sample DAE Problem

In the EASY-FIT DAE parameter estimation problem BOND, we wish to wish to fit simulated data that model the transition of a photon in a hydrogen-hydrogen bond [34]. The model functions (4.10) are evaluated at $M = m_t m_{Exp} = 24$ points, and a uniformly distributed error of 5% is added to the function values using parameter values $\mathbf{p} = (8.43, 0.29, 2.46, 8.76 \times 10^4)^\mathsf{T}$. The unknown parameter vector is $\mathbf{p} = (p_1, p_2, p_3, p_4)^\mathsf{T}$. The model functions

$$h_1(\mathbf{p}; \mathbf{x}(\mathbf{p}; t), \mathbf{y}(\mathbf{p}, t)) = x_1, \tag{4.7}$$

$$h_2(\mathbf{p}; \mathbf{x}(\mathbf{p}; t), \mathbf{y}(\mathbf{p}, t)) = x_2, \tag{4.8}$$

$$h_3(\mathbf{p}; \mathbf{x}(\mathbf{p}; t), \mathbf{y}(\mathbf{p}, t)) = y, \tag{4.9}$$

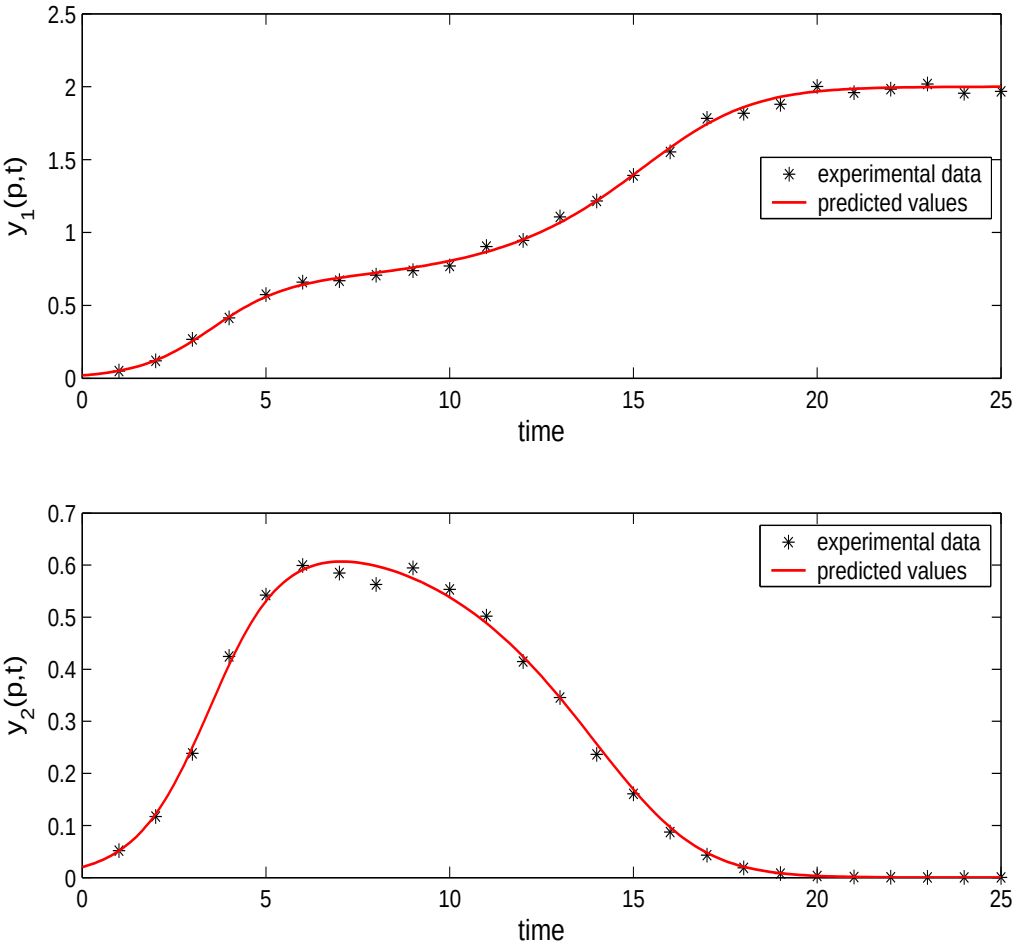


Figure 4.2: Plot of experimental data versus theoretically predicted model values obtained from $\mathbb{SOCS}$ for ODE parameter estimation problem COMPET.

depend on the solution $\mathbf{x}(\mathbf{p};t)$ and $\mathbf{y}(\mathbf{p};t)$ of a system of 2 ODEs and 1 algebraic equation

$$\begin{aligned} \dot{x}_1(\mathbf{p};\mathbf{x},\mathbf{y},t) &= -k_1 x_1 + p_2 y \ , \quad x_1(0) = 0, \\ \dot{x}_2(\mathbf{p};\mathbf{x},\mathbf{y},t) &= -k_4 x_1 + p_3 y \ , \quad x_2(0) = 1, \\ 0 &= k_1 x_1 + k_4 x_2 - (p_2 + p_3) y \ , \quad y(0) = 0, \end{aligned} \tag{4.10}$$

where $k_1 = p_1 \cdot 1 \times 10^{-10}$ and $k_4 = p_4 \cdot 1 \times 10^{-10}$. The initial guess for the parameters is $\mathbf{p}_0 = (10, 0.5, 5, 1 \times 10^5)^\mathsf{T}$ with lower and upper bounds given by $0 \leq \mathbf{p} \leq 1 \times 10^6$. The least-squares data-fitting problem is

$$\begin{aligned} &\min_{\mathbf{p}} \ \sum_{k=1}^{m_{Exp}} \sum_{i=1}^{m_t} (h_k(\mathbf{p};\mathbf{x}(\mathbf{p};t_i),\mathbf{y}(\mathbf{p};t_i),t_i) - \hat{h}_i^k)^2, \\ &0 \leq \mathbf{p} \leq 1 \times 10^6, \ \mathbf{p} \in \mathbb{R}^4. \end{aligned} \tag{4.11}$$

This problem is solved using the default trapezoidal discretization method and the linear initial guess method for the dynamic variables given by (3.2). The solution is summarized in Table 4.4. In the $\mathbb{SOCS}$ solution, parameter p_1 reached its lower bound of 0. EASY-FIT and $\mathbb{SOCS}$ have clearly converged to different solutions based on the value of p_1. However, the objective function values still agree to 3 decimal places. This is an example where each software package has converged to a different locally optimal solution. In this case, $\mathbb{SOCS}$ finds a better solution than EASY-FIT. A plot of the experimental data and $\mathbb{SOCS}$ solution to this DAE parameter estimation problem is given in Figure 4.3.

Problem	$\mathbf{p}$ and obj. values	EASY-FIT	$\mathbb{SOCS}$
BOND	p_1	2.364×10^5	0
	p_2	0.307	0.297
	p_3	2.494	2.489
	p_4	8.780×10^4	8.765×10^4
	obj.	7.617×10^{-4}	7.233×10^{-4}

Table 4.4: Translator results obtained from EASY-FIT and $\mathbb{SOCS}$ for DAE model BOND.

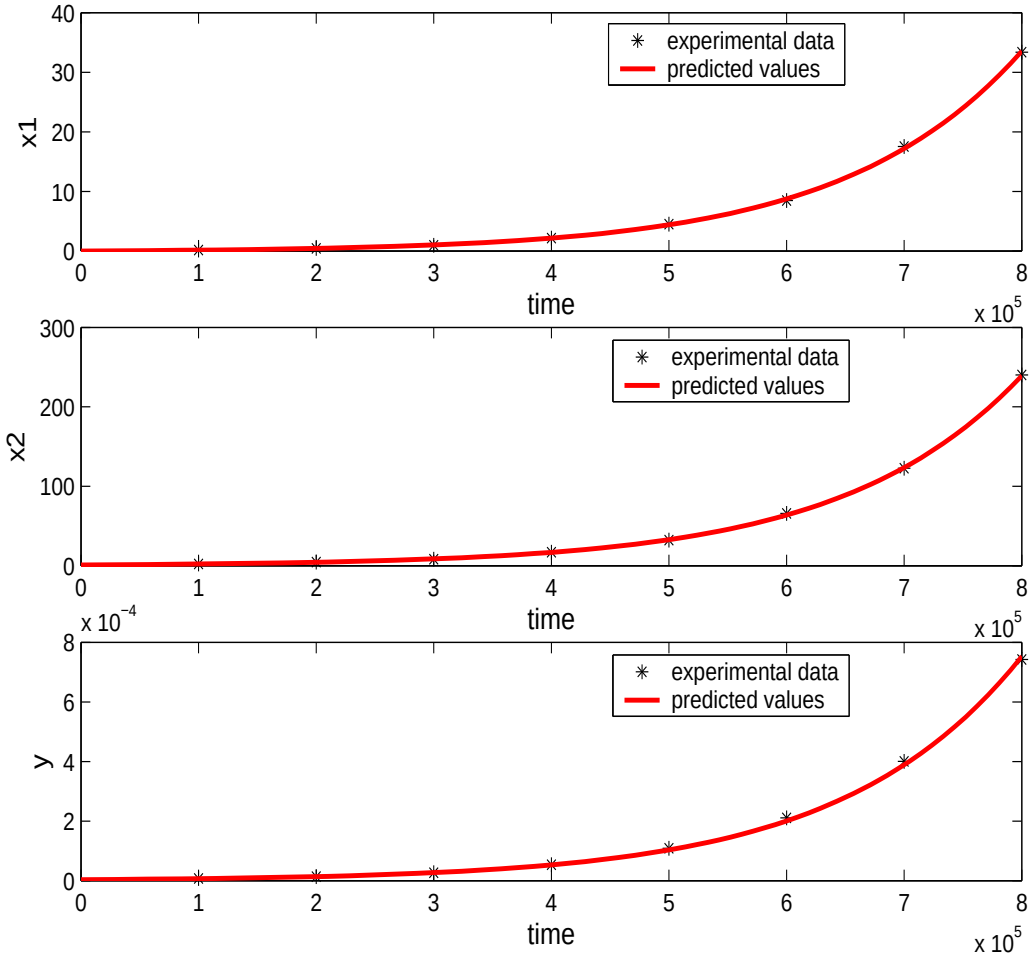


Figure 4.3: Plot of experimental data versus theoretically predicted model values obtained from SOCS for DAE parameter estimation problem BOND.

4.4 Discussion

We selected the 3 problems described above because they belong to the set of problems in which $\mathbb{SOCS}$ and EASY-FIT converge to a solution with approximately the same objective function value. The explicit and DAE models also illustrate the potential for locally optimal solutions in parameter estimation problems. EF2$\mathbb{SOCS}$ correctly translates these 3 problems; however there are some problems in the EASY-FIT database that EF2$\mathbb{SOCS}$ cannot translate. We now summarize the 5 subsets of problems that we have identified that EF2$\mathbb{SOCS}$ cannot translate.

Type I: *Models with discontinuities in their right-hand side functions.* These discontinuties may cause slow convergence, very small step size selection, or even divergence in the numerical methods used by $\mathbb{SOCS}$. Although it is possible for $\mathbb{SOCS}$ to solve problems with discontinuities, it is best to avoid discontinuous behaviour. Several suggestions as to how to do so are given in [5].

Type II: *Models with multiple phases and constraints applied at interior points of a phase*, i.e., at non-boundary points. These models are supported by $\mathbb{SOCS}$; however, because the treatment of these constraints in $\mathbb{SOCS}$ typically requires analysis on a problem-by-problem basis, it is not possible to automate the translation process. For example, the required $\mathbb{SOCS}$ code could depend on quantities such as concentration variables, breakpoints, and initial values involving unknown parameters. Note that single-phase models with constraints applied at interior points of a phase can be translated by EF2$\mathbb{SOCS}$. A breakpoint is inserted at the point at which the constraint is applied, forming an equivalent multi-phase model but with the constraint applied at the end of the phase. Because the is required analysis on a problem-by-problem basis, multi-phase models with constraints applied at interior points of a phase cannot be translated by EF2$\mathbb{SOCS}$. However, these problems can be translated by EF2$\mathbb{SOCS}$ if the user defines the models in PCOMP so that the constraints are applied at boundary points of a phase. This is done by defining a breakpoint to coincide with the point at which the constraint is applied so that the constraint is applied at the end of the phase. The state

variables should be continuous across these breakpoints because they were continuous in the single-phase model.

Type III: *Models with discontinuous solutions across breakpoints with initial values that depend on an unknown parameter or state variable.* These models are supported by SOCS; however, as with Type II, the treatment of the required constraints in SOCS typically requires analysis on a problem-by-problem basis because the SOCS code could depend on quantities such as concentration variables, breakpoints, and initial values involving unknown parameters. Therefore, it is not possible to automate the translation process.

Type IV: *Models with variable breakpoints*, i.e., breakpoints that are treated as parameters to be optimized. These models are not supported by SOCS and therefore cannot be translated by EF2SOCS.

Type V: *High-index DAEs*, i.e., index higher than 1. These models are not directly supported by SOCS and therefore cannot be translated by EF2SOCS; see Section 2.2.3.

For all problems except those of Type I, EF2SOCS outputs a warning. Warnings from Type III problems require examination of PCOMP code to recognize the potential difficulties that EF2SOCS may have encountered during translation. A thorough check by EF2SOCS is not performed; however the straightforward cases where the exact state variable name and unknown parameter name are used for a discontinuity across a breakpoint would generate a warning. EF2SOCS does not check the dependence of one variable on another to resolve the underlying value. So, for example, if a local variable `v` is set to be the value of the state variable `y` and `v` is used in the initial value of a breakpoint that is classified as Type III, EF2SOCS would not generate a warning. If the name `y` is used, then EF2SOCS recognizes the name, and if the initial value is classified as Type III, EF2SOCS then generates a warning. Warnings generated from Type II, Type IV, and Type V problems do not require examination of PCOMP code. The file containing the problem-specific constants, e.g., see Appendix A, provides sufficient information about the model to recognize problems due to these 3 types.

The number of problems of each type for all three categories of models is given in Table 4.5. Excluding these problems that cannot be translated, we summarize the results of running EF2SOCS on the remaining 691 problems from the EASY-FIT database in Table 4.6.

Model	Type I	Type II	Type III	Type IV	Type V
Exp.	0	0	N/A	N/A	N/A
ODEs	28	22	34	28	N/A
DAEs	13	3	4	1	1

Table 4.5: Summary of number of problems that cannot be translated by EF2SOCS.

Model	# Probs	EF2SOCS warnings	FORTRAN90 errors	converged solutions
Exp.	231	0.0 %	0.0 %	(231) 100.0 %
ODEs	436	0.0 %	0.0 %	(429) 98.4 %
DAEs	24	0.0 %	0.0 %	(24) 100.0 %

Table 4.6: Testing results obtained from EF2SOCS and SOCS.

Considering the 231 explicit models, none of the problems produced an EF2SOCS warning or a FORTRAN90 compiling error, and SOCS converged to a solution for every problem. Similarly, for the 24 DAE models, none of the problems produced an EF2SOCS warning or a FORTRAN90 compiling error, and SOCS converged to a solution for every problem. For the 436 ODE models, none of the problems produced an EF2SOCS warning or a FORTRAN90 compiling error; however, there were 7 models that were correctly translated but could not be solved by SOCS. We now give an explanation of the process of solving a parameter estimation problem in SOCS using EF2SOCS.

The process of using EF2SOCS to solve a parameter estimation problem using SOCS involves running EF2SOCS with two files: The first contains the model functions defined in PCOMP and the second contains the problem-specific constants. EF2SOCS generates a warning if the PCOMP

code does not follow the requirements given in Section 3.1 or if the problem is one of the 5 types of problems that cannot be translated. A warning prevents EF2$\mathbb{SOCS}$ output from compiling because it is printed at the beginning of the file containing the generated FORTRAN90 $\mathbb{SOCS}$ code. It is possible that EF2$\mathbb{SOCS}$ output would compile by simply removing any translator warnings from this file. If EF2$\mathbb{SOCS}$ output is able to be compiled, the executable is then run. If a solution is given by $\mathbb{SOCS}$, we then check if $\mathbb{SOCS}$ converged to this solution because it is possible that $\mathbb{SOCS}$ gives a solution that has not converged.

To obtain converged solutions in $\mathbb{SOCS}$, various initial guess strategies were applied to the dynamic variables and the unknown parameters. We group the initial guess strategies into 3 stages. The stages are designed to progressively give more information about the solution obtained by EASY-FIT. If $\mathbb{SOCS}$ fails to converge to a solution in Stage 1, we proceed to Stage 2. Similarly, if $\mathbb{SOCS}$ fails to converge to a solution in Stage 2, we proceed to Stage 3. Stage 1 consists of 2 related initial guesses, namely the EASY-FIT initial guess for the unknown parameters but 2 different choices for the parameter bounds. For the purposes of $\mathbb{SOCS}$ input, the parameter bounds p_{lb} and p_{ub} are either set to those used by EASY-FIT, or they are restricted according to the following formulas

$$p_{lb,i} = \begin{cases} p_{L,i}, & \text{if } |p_{L,i}| \leq 10, \quad i = 1, \ldots, n_u, \\ \text{not specified}, & \text{if } |p_{L,i}| > 10, \quad i = 1, \ldots, n_u, \end{cases}$$

$$p_{ub,i} = \begin{cases} p_{U,i}, & \text{if } |p_{U,i}| \leq 10, \quad i = 1, \ldots, n_u, \\ \text{not specified}, & \text{if } |p_{U,i}| > 10, \quad i = 1, \ldots, n_u, \end{cases} \tag{4.12}$$

where $p_{lb,i}$ represents the ith component of $\mathbf{p}_{lb}$ in $\mathbb{SOCS}$ and similarly for $p_{ub,i}$. Also, $p_{L,i}$ represents the component i of $\mathbf{p}_L$ in EASY-FIT and similarly for $p_{U,i}$. In general, for the EASY-FIT database it seems reasonable to assume that an upper bound of at most 10 (or a lower bound of at most -10) on an unknown parameter is sensible; i.e., the value is based on an actual bound and not a fictitious value. We make this choice to restrict the bounds because it became apparent that an upper bound of 10^6, which is commonly used in EASY-FIT, causes poor scaling within the numerical algorithms

of $\mathbb{SOCS}$ when the magnitude of the optimal parameter value is not this large. In equations (4.12), if the parameter bounds are not defined, then we let $\mathbb{SOCS}$ compute bounds for the unknown parameters internally.

In $\mathbb{SOCS}$, the optimal control problem variables and constraints are scaled in an attempt to improve the conditioning and efficiency of the underlying NLP problem. Variable scaling is performed by estimating the smallest and largest variable values from the user-supplied variable bounds. If bounds are not supplied, they are computed internally in an attempt to produce a well-scaled Jacobian, e.g., by normalizing the rows and columns of the Jacobian to have the same magnitude, along with several other heuristics; see [7]. Effectively, if there are no definite bounds on the unknown parameters, it is better to not specify them when using $\mathbb{SOCS}$.

Similarly, Stage 2 uses the EASY-FIT optimal solution for the unknown parameters and the same 2 choices as Stage 1 for the parameter bounds. Stage 3 uses the EASY-FIT optimal solution for the unknown parameters but sets the bounds on the unknown parameters to be very close to the EASY-FIT optimal solution. Each stage also uses 4 different methods for constructing an initial guess for the dynamic variables $\mathbf{d}(t)$, given by (2.1). The method is defined using the array INIT, as described in Section 3.2. Next, we describe in detail each stage and its corresponding strategies.

We do not have access to the EASY-FIT initial guess for the dynamic variables, but we do have access to the initial guess and bounds used for the unknown parameters. Therefore our best attempt at using the same initial guess as EASY-FIT uses the same initial guess for the unknown parameters and the same unknown parameter bounds as used by EASY-FIT. We also use the default initial guess method for the dynamic variables in $\mathbb{SOCS}$; i.e., we linearly interpolate the boundary conditions using INIT(1)=1. If this does not lead to a converged solution in $\mathbb{SOCS}$, the first modification is to restrict the parameter bounds by removing non-sensible bounds used by EASY-FIT and continue using the default initial guess method for the dynamic variables in $\mathbb{SOCS}$.

We follow this strategy by using different initial guess methods for the dynamic variables in $\mathbb{SOCS}$. We try the linear interpolation of measurement data method INIT(1)=9, followed by the linear control approximation INIT(1)=6 using the Runge–Kutta–Taylor integrator, and finally the

linear control approximation INIT(1)=6 and the stiff BDF integrator INIT(2)=3. For each of these 3 methods, we first use the same unknown parameter bounds as EASY-FIT, and then the bounds are restricted. This combination of 4 guess methods for the dynamic variables, and either restricting or not restricting the parameter bounds, forms Stage 1.

Stage 2 consists of attempting to obtain a converged solution in $\mathbb{SOCS}$ by using the optimal EASY-FIT parameter values $\mathbf{p}^*$ as the initial guess for the unknown parameters in $\mathbb{SOCS}$. These values are a solution to the parameter estimation problem in EASY-FIT, and because we are trying to solve the same problems with $\mathbb{SOCS}$, they should also be close to an approximate solution to the problem in $\mathbb{SOCS}$. Again, we begin by using the same parameter bounds as EASY-FIT, followed by restricting the parameter bounds. The 4 different initial guess methods for the dynamic variables used in Stage 1 are used in the same order for Stage 2.

At this point, if $\mathbb{SOCS}$ has failed to converge to a solution we proceed to Stage 3. This stage consists of using the optimal EASY-FIT parameter values as the initial guess for the unknown parameters, but we also set the bounds on these to be $\pm 0.1\%$ of the EASY-FIT optimal values. We note that we cannot set the bounds to be exactly the EASY-FIT optimal values because this removes the degrees of freedom associated with the unknown parameters and may lead to an ill-posed problem. For example, if there are 2 unknown parameters declared in $\mathbb{SOCS}$ and both are fixed to specific values, i.e., the lower bound is equal to the upper bound, then there are no parameter values to be optimized. We find that $\pm 0.1\%$ is a safe range to use because a smaller value often causes the difference between the upper and lower parameter bounds to be smaller than the allowable $\mathbb{SOCS}$ constraint tolerance $\texttt{CONTOL} = 1.49 \times 10^{-8}$, essentially fixing the value of the parameter. In equations (4.13), we summarize all possible cases for determining the unknown parameter bounds when using Stage 3 strategies. These cases prevent bounds that in effect fix the parameter value because the optimal parameter value is less (in absolute value) than `CONTOL`.

$$p_{lb,i} = \begin{cases} p_i^* + p_i^* \cdot 0.001, & \text{if } p_i^* \leq -\texttt{CONTOL}, & i = 1, \ldots, n_u, \\ p_i^* - p_i^* \cdot 0.001, & \text{if } p_i^* \geq \texttt{CONTOL}, & i = 1, \ldots, n_u, \\ 0, & \text{if } p_i^* \geq 0 \text{ and } p_i^* < \texttt{CONTOL}, & i = 1, \ldots, n_u, \\ p_i^* - p_i^* \cdot 0.001, & \text{if } p_i^* \leq 0 \text{ and } p_i^* > -\texttt{CONTOL}, & i = 1, \ldots, n_u, \end{cases} \tag{4.13}$$

$$p_{ub,i} = \begin{cases} p_i^* - p_i^* \cdot 0.001, & \text{if } p_i^* \leq -\texttt{CONTOL}, & i = 1, \ldots, n_u, \\ p_i^* + p_i^* \cdot 0.001, & \text{if } p_i^* \geq \texttt{CONTOL}, & i = 1, \ldots, n_u, \\ p_i^* + p_i^* \cdot 0.001, & \text{if } p_i^* \geq 0 \text{ and } p_i^* < \texttt{CONTOL}, & i = 1, \ldots, n_u, \\ 0, & \text{if } p_i^* \leq 0 \text{ and } p_i^* > -\texttt{CONTOL}, & i = 1, \ldots, n_u. \end{cases}$$

Using the above methods in the order described, we get the results given in Table 4.7. Note that only the strategy with a default initial guess for the dynamic variables and the EASY-FIT initial unknown parameter guess and bounds is run on all 691 problems. All subsequent strategies are only run on the remaining unconverged problems. Also, the percentages reported are all relative to the total number of problems (691).

It is clear that removing the non-sensible parameter bounds allows $\mathbb{SOCS}$ to solve many more problems than it does otherwise. In fact, across all models in Stage 1, between 80% and 400% more problems were solved. For the explicit models, ODEs and DAEs, respectively, 46.3%, 25.9%, and 20.8%, of the total number of problems were solved using the default initial guess method for the dynamic variables. If we extend this to restricted bounds in $\mathbb{SOCS}$, then 85.7%, 63.5%, and 37.5% of the explicit models, ODEs and DAEs, respectively, were solved. By the end of Stage 1, 96.1% of the explicit models were solved, 88.3% of the ODEs were solved, and 75.2% of the DAEs were solved. Across all model types, Stage 1 strategies solved over 90% ($625/691 \approx 90.4\%$) of the total number of models. After combining Stage 1 and Stage 2, 99.1% of the explicit models were solved, 95.0% of the ODEs were solved, and 91.8% of the DAEs were solved. Finally, Stage 3 strategies solved the remaining 0.9% of the explicit models, an additional 3.9% of the ODEs, and the remaining 8.2% of the DAEs. Once all problems for a model type were solved, the remaining

strategies were not used and this is indicated by using "– –" in Table 4.7. We note that 7 ODEs could not be solved using any of the methods listed in this table. We now explain why $\mathbb{SOCS}$ could not solve these ODE models.

The ODE models that could be translated EF2$\mathbb{SOCS}$ but were unable to produce converged solutions in $\mathbb{SOCS}$ may be grouped into three categories. The ODEs were either too stiff, ill-conditioned, or required a better initial guess technique. Each of these categories is described next.

There were 5 problems with ODEs that were too stiff for $\mathbb{SOCS}$ to solve, namely, COPPER, COPPER_D, KEPLER, REACHMECH, and STIFF_DE. One mathematical interpretation of *stiffness* is that the solution being sought varies slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to meet stability requirements [37]; i.e., the step size is chosen to maintain numerical stability rather than to meet accuracy requirements [26], [55]. A common observation in these 5 problems is that $\mathbb{SOCS}$ is unable to solve the system of ODEs even when the optimization step is removed. So, by fixing the unknown parameters to specific values and removing the discrete data, all optimization steps are eliminated. Despite this, $\mathbb{SOCS}$ could not solve the system of ODEs, thus leaving no hope to solve the optimization problem. It is unclear at present why these problems are too stiff for the BDF integrator to solve. It may be useful to try another solver that uses the collocation method, e.g., as implemented in COLDAE [3], to see if it can solve the ODEs.

There was 1 problem, called BCBPLUS, that was ill-conditioned. To compute the sparsity pattern of the right-hand side of the system of ODEs, $\mathbb{SOCS}$ perturbs the NLP variables, and then tries to evaluate the right-hand side functions. During this perturbation evaluation a "function error" is detected. This message is generated by $\mathbb{SOCS}$ when the magnitude of the right-hand side functions exceeds 10^{16}. This leads to two negative effects. First, $\mathbb{SOCS}$ cannot detect right-hand side sparsity and therefore assumes the right-hand side is dense, making the computation of the finite difference derivatives much more expensive. Second, the numerical values of the Jacobian of the right-hand side are very large, resulting in the failure of the NLP algorithms.

The final problem, called FEDBATCH, could be solved by $\mathbb{SOCS}$; however this required a special initial guess technique not currently supported by EF2$\mathbb{SOCS}$. This FEDBATCH is a 4-phase problem, and it was eventually solved using the method of *analytical continuation* [2]. In other words, it was too difficult to obtain a sufficiently accurate initial guess to the problem in a heuristic manner. Instead we solved an easier problem consisting of only one phase using the optimal parameter values of EASY-FIT and fixing tight bounds on these values. The solution at the end of phase 1 is then used as the guess for the dynamic variables at the start of phase 2. This procedure is then repeated until the time interval was expanded to include all 4 phases. This stepwise procedure exploits the restart mechanism in $\mathbb{SOCS}$ to solve a sequence of increasingly difficult problems. This approach also utilizes the $\mathbb{SOCS}$ subroutine OCSTAU to communicate the necessary initial conditions for the phases; see Section 3.2. The concept of analytical continuation is a powerful method for obtaining solutions to many problems where standard initial guess methods fail to provide a sufficiently accurate initial guess [13]. The ability to use analytical continuation would be a significant addition to the functionality of EF2$\mathbb{SOCS}$.

As a final comparison, we categorize the objective function values of the converged solutions produced by $\mathbb{SOCS}$ as compared to those of EASY-FIT. The objective function value is either greater than, "equal to", or less than the objective function value of EASY-FIT. We say the objective function values are "equal" if they differ by less than 10%. We deem this to be an acceptable amount in absolute terms considering the magnitude of most EASY-FIT objective function values is in the range 10^{-2} to 10^{-4}, thus leading to at least 2 matching decimal places between the two objective function values. The results are based on using all Stage 1 and Stage 2 strategies in Table 4.7. Stage 3 strategies are not included because they force the solution in $\mathbb{SOCS}$ to be approximately equal to that of EASY-FIT, and therefore the inclusion of Stage 3 would skew the results. The results are also recorded in the order presented in Table 4.7. Therefore, the solutions may not be the best that $\mathbb{SOCS}$ can achieve. For example, if $\mathbb{SOCS}$ converges to a solution using the default method for the dynamic variables, the initial unknown parameter values of EASY-FIT, and the parameter bounds of EASY-FIT, it is possible that using a more sophisticated initial guess method for the dynamic

	Guess for $\mathbf{d}$	Parameter guess and bounds	Exp.	ODEs	DAEs
Stage 1	INIT(1)=1	$\mathbf{p}_0$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	(107) 46.3 %	(113) 25.9 %	(5) 20.8 %
		$\mathbf{p}_0$ and restricted bounds	(91) 39.4 %	(160) 37.6 %	(4) 16.7 %
	INIT(1)=9	$\mathbf{p}_0$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	(7) 3.0 %	(16) 3.7 %	(2) 8.3 %
		$\mathbf{p}_0$ and restricted bounds	(17) 7.4 %	(35) 8.0 %	(3) 12.5 %
	INIT(1)=6	$\mathbf{p}_0$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	N/A	(12) 2.8 %	(1) 4.2 %
		$\mathbf{p}_0$ and restricted bounds	N/A	(30) 6.9 %	(1) 4.2 %
	INIT(1)=6,	$\mathbf{p}_0$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	N/A	(6) 1.4 %	(1) 4.2 %
	INIT(2)=3	$\mathbf{p}_0$ and restricted bounds	N/A	(13) 3.1 %	(1) 4.2 %
Stage 2	INIT(1)=1	$\mathbf{p}^*$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	(3) 1.3 %	(6) 1.4 %	0
		$\mathbf{p}^*$ and restricted bounds	(3) 1.3 %	(3) 0.7 %	(1) 4.2 %
	INIT(1)=9	$\mathbf{p}^*$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	0	(1) 0.2 %	0
		$\mathbf{p}^*$ and restricted bounds	(1) 0.4 %	(6) 1.4 %	0
	INIT(1)=6	$\mathbf{p}^*$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	N/A	(2) 0.5 %	0
		$\mathbf{p}^*$ and restricted bounds	N/A	(7) 1.6 %	0
	INIT(1)=6,	$\mathbf{p}^*$ and $\mathbf{p}_{lb}, \mathbf{p}_{ub}$	N/A	0	0
	INIT(2)=3	$\mathbf{p}^*$ and restricted bounds	N/A	(4) 0.9 %	(3) 12.4 %
Stage 3	INIT(1)=1	$\mathbf{p}^*$ and $\mathbf{p}_{lb}$, $\mathbf{p}_{ub}$ as in (4.13)	(2) 0.9 %	(3) 0.7 %	(2) 8.2 %
	INIT(1)=9	$\mathbf{p}^*$ and $\mathbf{p}_{lb}$, $\mathbf{p}_{ub}$ as in (4.13)	– –	(1) 0.2 %	– –
	INIT(1)=6	$\mathbf{p}^*$ and $\mathbf{p}_{lb}$, $\mathbf{p}_{ub}$ as in (4.13)	N/A	(4) 0.9 %	– –
	INIT(1)=6,	$\mathbf{p}^*$ and $\mathbf{p}_{lb}$, $\mathbf{p}_{ub}$ as in (4.13)	N/A	(6) 2.1 %	– –
	INIT(2)=3				

Table 4.7: Percentage of converged solutions for various initial guess strategies.

variables could produce a solution with a lower objective function value. The same may be said of EASY-FIT; however it is suspected, and in some cases confirmed, that EASY-FIT is tuned to solve certain problems by using non-default values. For example, many problems use different tolerance values. Also, integration methods used to solve the system of differential equations varies from problem to problem.

For all three model types, over half of the problems converged to a solution with an objective function value at least as small as the one given by EASY-FIT. The percentages are 70.1%, 59.6%, and 66.7% for the explicit models, ODEs, and DAEs, respectively. It is relatively easier to solve the explicit models because they do not involve the additional complication of solving differential equations. This may account for the greater percentage of problems with an objective function value at least as small as that of EASY-FIT. On the other hand, 63.5% of the ODEs converged using the default initial guess method for the state variables; see Table 4.7. There is a significantly larger percentage of ODE problems than explicit models with an objective function value greater than that of EASY-FIT, perhaps suggesting that $\mathbb{SOCS}$ is converging to a sub-optimal solution when using the default initial guess method. It may be informative to restrict the initial guess method to one that is more advanced, e.g., linear interpolation of the experimental data or solution of an IVP with a linear control approximation using the Runge–Kutta–Taylor integrator, and compare the differences in objective function values. It is clear for all three model types that many locally optimal solutions exist. Therefore, it would also be informative to use global optimization software to verify the current solutions as being global or locate the global solutions to these problems. Note that almost a dozen explicit models were coded using the global optimization software package GlobSol [31]. Most of these problems were run for several days using a 1.33GHz PowerPC G4 processor with 768MB of RAM. and failed to converge over that time. Others were run for 2 weeks and also failed to converge over that time, reinforcing the claim made in Chapter 2 that current computational methods for solving global optimization problems generally require large amounts of computation time.

Model	smaller objective value	equal objective value	larger objective value
Exp.	(52) 22.5 %	(110) 47.6 %	(69) 29.9 %
ODEs	(45) 10.3 %	(215) 49.3 %	(169) 38.8 %
DAEs	(5) 20.9 %	(11) 45.8 %	(8) 33.3 %

Table 4.8: Comparison of $\mathbb{SOCS}$ objective function values to EASY-FIT objective function values.

Overall we find that imposing restricted bounds on the unknown parameters greatly increases the ability of $\mathbb{SOCS}$ to converge to a solution. Also, with a sufficiently good initial guess for the dynamic variables, over 90% of the total number of problems could be solved. Just under 70% of the total number of problems could be solved by using the default initial guess method for the dynamic variables, showing that the default method is indeed effective. When considering the problems that could be translated but not solved, the main difficulty is that the ODEs were too stiff to solve. It is unclear whether $\mathbb{SOCS}$ would benefit from another integrator for very stiff problems. Finally, it is apparent that numerous parameter estimation problems in the EASY-FIT database have many locally optimal solutions, suggesting the need for global optimization functionality.

Chapter 5

Conclusions and Future Work

EASY-FIT and $\mathbb{SOCS}$ are two software packages used to solve parameter estimation problems. We described these packages and gave a survey of other available optimal control software packages. To facilitate the use of solving parameter estimation problems in $\mathbb{SOCS}$ and at the same time test $\mathbb{SOCS}$ on a large test set of problems, we designed and implemented a source-to-source translator called EF2$\mathbb{SOCS}$ to translate the input for EASY-FIT into the corresponding input for $\mathbb{SOCS}$. We described the design of EF2$\mathbb{SOCS}$, including its GUI. We also described the effectiveness of the translation in terms of comparing the solutions produced by both software packages. Finally, we suggested some possible improvements for both EF2$\mathbb{SOCS}$ and $\mathbb{SOCS}$.

We proposed a 3-stage approach to produce converged solutions in $\mathbb{SOCS}$. Each stage was composed of various initial guess strategies for the dynamic variables and unknown parameters. The stages were designed to use an initial guess that is progressively closer to the optimal solution found by EASY-FIT.

Using this 3-stage approach we were able to solve all 231 explicit models in the EASY-FIT database using $\mathbb{SOCS}$. There were 112 ODEs and 22 DAEs that could not be translated by EF2$\mathbb{SOCS}$ either because the functionality was not supported in $\mathbb{SOCS}$ or because the treatment of the constraints in $\mathbb{SOCS}$ typically required analysis on a problem-by-problem basis. The nature of this difficulty precluded automation of the translation process. From the remaining 436 ODEs, all but 7 were solved in $\mathbb{SOCS}$. An investigation into the reasons for the failure of $\mathbb{SOCS}$ on these 7 problems revealed that 5 problems were too stiff to solve, 1 problem was ill-conditioned and led to large numerical values in the Jacobian of the right-hand side and ultimately the failure of the NLP solvers, and 1 problem required the use of analytical continuation. All 24 remaining DAEs

were solved using SOCS.

For the problems that could be translated, we found that without making any use of the EASY-FIT solution, 96.1% of the explicit models could be solved, 88.3% of the ODEs could be solved, and 75.2% of the DAEs could be solved. These percentages account for over 90% of the 691 problems from the EASY-FIT database.

In the process of solving these 691 problems from the EASY-FIT database, we found that if there are no physically motivated bounds on the unknown parameters it is better to leave them unspecified when using SOCS than to choose fictitious values. It is also clear that a large number of these problems have many locally optimal solutions, suggesting the need for global optimization functionality. Although the ability to use norms alternative to the L_2-norm was not required, this functionality, which is currently not available in SOCS, could be useful for solving parameter estimation problems.

Future work consists of attempting to solve the ODEs that were too stiff to solve in SOCS with another boundary-value problem solver that uses the collocation method, such as COLDAE [3]. In terms of additions to EF2SOCS, the ability to correctly translate high-index DAEs could be added as well as the ability to use analytical continuation as an initial guess method to solve parameter estimation problems in SOCS. It may also be helpful to have other users try EF2SOCS to gain useful feedback on its usability.

References

[1] U.M. Ascher and L.R. Petzold. Projected implicit runge-kutta methods for differential- algebraic equations. *SIAM Journal on Mathematical Analysis*, 28(4):1097–1120, 1991.

[2] U.M. Ascher and L.R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[3] U.M. Ascher and R.J. Spiteri. Collocation software for boundary value differential-algebraic equations. *SIAM Journal on Scientific Computing*, 15(4):938–952, 1994.

[4] E. Beltrami. *Mathematics for Dynamic Modeling*. Academic Press, Inc., Boston, MA, USA, 1987.

[5] J.T. Betts. *Practical methods for optimal control using nonlinear programming*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[6] J.T. Betts and S.L Campbell. Discretize then optimize. In D. R. Ferguson and T. J. Peters, editors, *Mathematics for Industry: Challenges and Frontiers*, pages 140–157, SIAM, Philadelphia, 2005.

[7] J.T. Betts and W.P. Huffman. Sparse Optimal Control Software $\mathbb{SOCS}$. Technical Report Mathematics and Engineering Analysis MEA-LR-085, Boeing Information and Support Services, The Boeing Company, P.O. Box 3707, Seattle, WA 98124-2207, July, 1997.

[8] C.H. Bischof, A. Carle, G.F. Corliss, and A. Griewank. ADIFOR: Automatic differentiation in a source translation environment. In Paul S. Wang, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 294–302, New York, 1992. ACM Press.

[9] A.E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Blaisdell, New York, 1969.

[10] A.E. Bryson Jr. Optimal control – 1950 to 1985. *IEEE Control Systems Magazine*, 16(3):26–33, 1996.

[11] R. Bulirsch, E. Nerz, H.J. Pesch, and O. von Stryk. Combining direct and indirect methods in optimal control: range maximization of a hang glider. In R. Bulirsch, A. Miele, J. Stoer, and K.-H. Well, editors, *Optimal Control - Calculus of Variations, Optimal Control Theory and Numerical Methods, International Series of Numerical Mathematics*, volume 111, pages 273–288. Basel: Birkhäuser, 1993.

[12] R.L. Burden and J.D. Faires. *Numerical Analysis*. Brooks–Cole, Boston, MA, USA, eighth edition, 2004.

[13] H.T. Davis. *Introduction to Nonlinear Differential and Integral Equations*. Dover, New York, 1962.

[14] M. Dobmann, M. Liepelt, K. Schittkowski, and C. Traßl. PCOMP: A Fortran code for automatic differentiation, language description and users guide. Technical report, Dept. of Mathematics, University of Bayreuth, Germany, 1995.

[15] EASY-FIT. Software for parameter estimation, 2007. http://www.old.uni-bayreuth.de/departments/math/~kschittkowski/easy_fit.htm.

[16] W.R. Esposito and C.A. Floudas. Deterministic global optimization in nonlinear optimal control problems. *Journal of Global Optimization*, 17(1-4):97–126, 2000.

[17] M.E. Fisher and L.S. Jennings. Discrete-time optimal control problems with general constraints. *ACM Transactions on Mathematical Software*, 18(4):401–413, 1992.

[18] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, second edition, 1987.

[19] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[20] C.W. Gear. The automatic integration of stiff ordinary differential equations. *IFIP Congress (1)*, pages 187–193, 1969.

[21] C.W. Gear. Differential algebraic equations, indices, and integral algebraic equations. *SIAM Journal on Numerical Analysis*, 27(6):1527–1534, 1990.

[22] P.E. Gill, W. Murray, and M.A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. Technical Report NA–97–2, San Diego, CA, 1997.

[23] P.E. Gill, W. Murray, M.A. Saunders, and M.H. Wright. User's guide for SOL/NPSOL: a Fortran package for nonlinear programming. Technical Report SOL 83-12, Department of Operations Research, Standford University, California, 1983.

[24] C.J. Goh, C.M. Wang, and K.L. Teo. Unified approach for structural optimization revisited: variable segment boundaries and variable interior point constraints. *Structural Optimization*, 3:133–140, 1991.

[25] E. Hairer, C. Lubich, and M. Roche. *The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods*, volume 1409 of *Lecture Notes in Mathematics*. Springer-Verlag, New York, 1989.

[26] E. Hairer and G. Wanner. *Solving ordinary differential equations II: Stiff and differential-algebraic problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, Germany, 1991.

[27] F.B. Hanson and D. Ryan. Optimal harvesting with both population and price dynamics. *Mathematical Biosciences*, 148(2):129–146, 1998.

[28] L.S. Jennings, M.E. Fisher, K.L. Teo, and C.J. Goh. MISER3: solving optimal control problems–an update. *Advances in Engineering Software*, 13(4):190–196, 1991.

[29] L.S. Jennings, M.E. Fisher, K.L. Teo, and C.J. Goh. MISER3: software for solving optimal control problems, 2005. http://www.cado.uwa.edu.au/miser/.

[30] H.R. Joshi. Optimal control of an HIV immunology model. *Optimal Control Applications and Methods*, 23(4):199–213, 2002.

[31] R.B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1996.

[32] H.B. Keller. *Numerical Methods for Two-Point Boundary Value Problems*. Blaisdell, Waltham, MA, London, 1968.

[33] P. Kruchten. Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[34] L. Lapidus, R.C. Aiken, and Y.A. Liu. *The occurrence and numerical solution of physical and chemical systems having widely varying time constants.* In: Willoughby E.A. (ed.): Stiff Differential Systems. Plenum Press, New York, USA, 1974.

[35] S. Li and L.R. Petzold. Design of new daspk for sensitivity analysis. Technical report, University of California at Santa Barbara, Santa Barbara, CA, USA, 1999.

[36] L. Liberty and S. Kucherenko. Comparison of deterministic and stochastic approaches to global optimization. *International Transactions in Operations Research*, 12(3), 2005.

[37] C.B. Moler. *Numerical Computing with Matlab.* Society for Industrial and Applied Mathematics, 2004.

[38] Sparse Optimal Control Software $\mathbb{SOCS}$. Mathematics and computing technology, 2008. `http://www.boeing.com/phantom/socs/applications.html`.

[39] P.M. Pardalos, D. Shalloway, and G.L. Xue. Optimization methods for computing global minima of non-convex potential energy functions. *Journal of Global Optimization*, 4:117–133, 1994.

[40] L.S. Pontryagin. *The Mathematical Theory of Optimal Processes.* Wiley-Interscience, New York, NY, USA, 1962.

[41] L.L. Raja, R.J. Kee, R. Serban, and L.R. Petzold. Dynamic optimization of chemically reacting stagnation flows. In M.D. Allendorf, M.R. Zachariah, L. Mountziaris, and A.H. McDaniel, editors, *Fundamental Gas-phase and Surface Chemistry of Vapor-Phase Materials Synthesis*, pages 340–351. Soc. Proc. Series 98-23, 1999.

[42] J.R. Rice and R.F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computational Science & Engineering*, 3(3):44–53, 1996.

[43] I.M. Ross and F. Fahroo. A perspective on methods for trajectory optimization. In *Proceedings of the AIAA/AAS Astrodynamics Specialist Conference*, Monterey, CA, August 2002. AIAA Paper No. 2002-4727.

[44] K. Schittkowski. NLPQL: a FORTRAN subroutine solving constrained nonlinear programming problems. *Annals of Operations Research*, 5:485–500, 1985.

[45] K. Schittkowski. *Numerical Data Fitting in Dynamical Systems: A Practical Introduction with Applications and Software.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[46] K. Schittkowski. private communication, May 2007.

[47] J. Schropp. Projected Runge-Kutta methods for differential algebraic equations of index 3. *Konstanzer Schriften in Mathematik und Informatik*, 2003.

[48] A. Schwartz. Theory and implementation of numerical methods based on runge-kutta integration for solving optimal control problems. Ph.D. Dissertation, Dept. of Electrical Engineering, University of California, Berkeley, 1996. `http://robotics.eecs.berkeley.edu/~adams`.

[49] A. Schwartz and E. Polak. On the use of consistent approximations in the solution of semi-infinite optimization and optimal control problems. *Mathematical Programming*, 62:385–415, 1993.

[50] A. Schwartz and E. Polak. Consistent approximations for optimal control problems based on Runge–Kutta integration. *SIAM Journal on Control and Optimization*, 34(4), 1996.

[51] A. Schwartz, E. Polak, and Y.Q. Chen. RIOTS: a MATLAB toolbox for solving optimal control problems, 1997. `http://www.schwartz-home.com/~adam/RIOTS/`.

[52] R. Serban, W.S. Koon, M. Lo, J.E. Marsden, L.R. Petzold, S.D. Ross, and R. Wilson. Halo orbit mission correction maneuvers using optimal control. *Automatica*, 38(4):571–583, 2002.

[53] R. Serban and L.R. Petzold. COOPT – a software package for optimal control of large-scale differential-algebraic equation systems. *Mathematics and Computers in Simulation*, 56(2):187–203, 2001.

[54] H. Seywald and E.M. Cliff. Goddard problem in presence of a dynamic pressure limit. *J. Guidance, Control and Dynamics*, 16(4):776–781, 1993.

[55] L.F. Shampine. *Numerical Solution of Ordinary Differential Equations*. Chapman and Hall, London, 1994.

[56] T. Slawig. Algorithmic sensitivity analysis in the climate model *climber 2*. Technical Report 29, Technische Universität Berlin, Institut für Mathematik, 2006.

[57] O. von Stryk. Numerical solution of optimal control problems by direct collocation. In *Optimal Control Calculus of Variations, Optimal Control Theory and Numerical Methods, International Series of Numerical Mathematics*, volume 111, pages 129–143, 1993.

[58] O. von Stryk. DIRCOL: a direct collocation method for the numerical solution of optimal control problems, 1999. `http://www.sim.tu-darmstadt.de/sw/dircol/dircol_app.html`.

[59] O. von Stryk and R. Bulirsch. Direct and indirect methods for trajectory optimization. *Annals of Operations Research*, 37:357–373, 1992.

[60] J. Xu, J.J. Heys, V.H. Barocas, and T.W. Randolph. Permeability and diffusion in vitreous humor: implications for drug delivery. *Phamaceutical Research*, 18:664–669, 2000.

Appendix A

Problem Data for Model TP333

```
1. problems\TP333
2. TP333         1
3. Exponential data fitting
4. Demo
5. Schittkowski
6. Experimental
7. Null
8. Null
9. t
10. NPAR  =     003    0  000
11. NRES  =     000
12. NEQU  =     000
13.
14. NODE  =       0
15. NCONC =     000
16. NTIME =    0008    0
17. NMEAS =     001
18. NPLOT =    0050
19. NOUT  =       0
20. METHOD=      01  000    2   -1
21. OPTP1 =   00110
22. OPTP2 =   00030
23. OPTP3 =      02
24. OPTE1 =   1.0000E-12
25. OPTE2 =   1.0000E-12
26. OPTE3 =   1.0000E+00
27. ODEP1 =       0
28. ODEP2 =       1    0    0    0   0
29. ODEP3 =      00
30. ODEP4 =       0
31. ODEE1 =   0.0
32. ODEE2 =   0.0
33. ODEE3 =   0.0
34.
x1               0.000000E+00        3.000000E+01        1.000000E+03
x2               0.000000E+00        4.000000E-02        1.000000E+03
x3               0.000000E+00        3.000000E+00        1.000000E+03
35. SCALE =      -1
36.
4.00000E+00 7.210000000000E+01 1.00E+00
5.75000E+00 6.560000000000E+01 1.00E+00
7.50000E+00 5.590000000000E+01 1.00E+00
2.40000E+01 1.710000000000E+01 1.00E+00
3.20000E+01 9.800000000000E+00 1.00E+00
4.80000E+01 4.500000000000E+00 1.00E+00
7.20000E+01 1.300000000000E+00 1.00E+00
9.60000E+01 6.000000000000E-01 1.00E+00
37. NLPIP         0
```

```
38. NLPMI         0
39. NLPAC         0.0
40. NDISCO=       0
y(t)
"Schittkowski K. (1987): More Test Examples for Nonlinear Programming,
Lecture Notes in Economics and Mathematical Systems, Vol. 282, Springer
Initial values:
3.00E+01 4.00E-02 3.00E+00"
```

Appendix B

SOCS Code for Problem TP333

```
      PROGRAM EXPLICIT_MODEL
C
C       ----THIS IS A PROGRAM FOR A PARAMETER ESTIMATION PROBLEM CHOSEN FROM
C           THE EASY-FIT SOFTWARE PACKAGE (TP333)
C
C       ******************************************************************
        IMPLICIT DOUBLE PRECISION  (A-H,O-Z)
        PARAMETER (MXIW=100000000,MXRW=100000000,MXC=10000000)
        PARAMETER (MXDP=3, MXPHS=1)
C
        COMMON /ODEWRK/ WORK(MXRW)
        COMMON /ODEIWK/ IWORK(MXIW)
        COMMON /ODESPL/ CSTAT(MXC)
        DIMENSION IPCPH(MXPHS+1),DPARM(MXDP),IPDPH(MXPHS+1)
        PARAMETER (MAXRWD=8, MAXCLD=3)
C
        EXTERNAL  DUMYPF,  DUMYPR,  EXPINP,  EXPRHS,  EXPDDL,  DUMYIG
C
C       ******************************************************************
C
C       ----WORKING ARRAYS USED BY HDSOPE ROUTINE
C
        NIWORK = MXIW
        NWORK = MXRW
        MAXCS = MXC
        MAXDP = MXDP
        MAXPHS = MXPHS
C
C       ******************************************************************
C
C       ----SETS EVERY OPTIONAL PARAMETER FOR THE SUBROUTINES HDSOCS AND
C           HDSOPE TO ITS DEFAULT VALUE
C
        CALL HHSOCS('DEFAULT')
C
C       ----MAXIMUM NUMBER OF DISCRETE DATA VALUES PER PHASE
C
        CALL HHSOCS('MXDATA=8')
C
C       ----MAXIMUM NUMBER OF PARAMETERS PER PHASE
C
        CALL HHSOCS('MXPARM=3')
C
C       ----MAXIMUM NUMBER OF STATES TO BE LOADED
C
        CALL HHSOCS('MXSTAT=4')
C
C       ----OPTIMAL CONTROL OUTPUT LEVEL (TERSE)
```

```
C
      CALL HHSOCS('IPGRD=5')
C
C     ----MAXIMUM NUMBER OF FUNCTION EVALUATIONS
C
      CALL HHSNLP('MAXNFE=50000')
C
C     ----MAXIMUM NUMBER OF ITERATIONS
C
      CALL HHSNLP('NITMAX=200')
C
C     ----OUTPUT UNIT NUMBER
C
      IPU = 3
C
C     ----CALL HDSOPE WITH THE APPROPRIATE ARGUMENTS FOR THE PROBLEM
C
      CALL HDSOPE(EXPINP,DUMYIG,EXPRHS,DUMYPF,DUMYPR,EXPDDL,
     &            IWORK,NIWORK,WORK,NWORK,MAXPHS,
     &            CSTAT,MAXCS,IPCPH,DPARM,MAXDP,IPDPH,NEEDED,IER)
C
      STOP
      END
C
C
      SUBROUTINE EXPRHS(IPHASE,T,YVEC,NYVEC,PARM,NPARM,FRHS,NRHS,
     & IFERR)
C
C     ----EVALUATE RIGHT HAND SIDE OF DATA FITTING FUNCTION(S)
C
C     ****************************************************************
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
      DIMENSION YVEC(NYVEC),FRHS(NRHS),PARM(NPARM)
C
      PARAMETER (NCONC=1)
      COMMON /CONPAR/ CONC(NCONC)
C
      DOUBLE PRECISION  H
C
      DOUBLE PRECISION  P1, P2, P3, T
C
C     ****************************************************************
C
C     ----INITIALIZE USER-DEFINED ERROR FLAG
C
      IFERR = 0
C
C     ----LOAD PARAMETER VECTOR
C
      P1 = PARM(1)
      P2 = PARM(2)
      P3 = PARM(3)
C
```

```
C     ----COMPUTE VALUES FOR FITTING FUNCTION(S)
C
      H = P1*EXP(-P2*T) + P3
C
      FRHS(1) = YVEC(1) - H
C
      RETURN
      END
C
C
      SUBROUTINE EXPINP(IPHASE,NPHS,METHOD,NSTG,NCF,NPF,NPV,NAV,NGRID,
     &                  INIT,MAXMIN,MXPARM,P0,PLB,PUB,PLBL,
     &                  MXSTAT,Y0,Y1,YLB,YUB,STSKL,STLBL,MXPCON,CLB,CUB,
     &                  CLBL,MXTERM,COEF,ITERM,TITLE,IER)
C
C     ----INITIALIZE DATA FITTING PROBLEM
C
C     ****************************************************************
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
C  ARGUMENTS:
      INTEGER    IPHASE,NPHS,METHOD,NSTG,NCF(5),NPF(2),NPV,NAV,NGRID,
     &           INIT(2),MAXMIN,MXPARM,MXSTAT,MXPCON,MXTERM,
     &           ITERM(4,MXTERM),IER
      DIMENSION  P0(MXPARM),PLB(MXPARM),PUB(MXPARM),Y0(0:MXSTAT),
     &           Y1(0:MXSTAT),YLB(-1:1,0:MXSTAT),YUB(-1:1,0:MXSTAT),
     &           STSKL(0:MXSTAT+MXPARM,2),CLB(MXPCON),CUB(MXPCON),
     &           COEF(MXTERM)
      CHARACTER  TITLE(3)*60,PLBL(MXPARM)*80,STLBL(0:MXSTAT)*80,
     &           CLBL(0:MXPCON)*80
C
      PARAMETER (MAXRWD=8,MAXCLD=3,NTIME=8)
      COMMON /EXPCM/ STDAT(MAXRWD,MAXCLD),NROWS
      PARAMETER (NCONC=1)
      COMMON /CONPAR/ CONC(NCONC)
C
      DOUBLE PRECISION  H
C
      DOUBLE PRECISION  P1, P2, P3, T
C
C     ****************************************************************
C
C     ----INITIALIZE USER-DEFINED ERROR FLAG
C
      IFERR = 0
C
C     ----NUMBER OF PHASES
C
      NPHS = 1
C
C     ----DEFINE INITIAL AND FINAL TIME
C
      TINITIAL = 4.0000000000D0
      TFINAL = 96.0000000000D0
```

```
C
C     ----NUMBER OF GRID POINTS
C
      NGRID = NTIME
C
C     ----SUCCESS/ERROR CODE
C
      IER = 0
C
      NTERM = 0
      NKON = 0
      TITLE(1) = 'PARAMETER ESTIMATION PROBLEM: TP333'
      TITLE(2) = 'SOLVED USING LEAST SQUARES'
      STLBL(0) = 'TIME    Time'
C
C     ----SET INTEGRATION METHOD TO TRAPEZOIDAL
C
      NSTG = 1
      METHOD = 3
C
C     ----INITIALIZE PROBLEM DATA
C
      CALL INIEXP
C
      PLBL(1) = 'p1       Parameter 1'
      PLBL(2) = 'p2       Parameter 2'
      PLBL(3) = 'p3       Parameter 3'
C
C     ----GUESS FOR INITIAL PARAMETER VALUES AND BOUNDS
C
      P0(1) = 3.0000000000D+01
      P0(2) = 4.0000000000D-02
      P0(3) = 3.0000000000D+00
      PLB(1) = 0.0000000000D0
      PUB(1) = 1000.0000000000D0
      PLB(2) = 0.0000000000D0
      PUB(2) = 1000.0000000000D0
      PLB(3) = 0.0000000000D0
      PUB(3) = 1000.0000000000D0
C
C     ----INITIALIZE PROBLEM DATA
C           NAV  - NUMBER OF ALGEBRAIC VARIABLES
C           NPV  - NUMBER OF DISCRETE PARAMETERS
C           NDE  - NUMBER OF DIFFERENTIAL EQUATIONS
C           NDAE - NUMBER OF DIFFERENTIAL ALGEBRAIC EQUATIONS
C           NDF  - NUMBER OF DATA FITTING FUNCTIONS
C
      NAV = 1
      NPV = 3
      NDE = 0
      NDAE = 0
      NDF = 1
C
C     ----NUMBER OF DIFFERENTIAL EQUATIONS
```

```
C
      NCF(1) = NDE
C
C     ----NUMBER ALGEBRAIC EQUATIONS
C
      NCF(2) = 0
C
C     ----NUMBER OF DISCRETE DATA FUNCTIONS
C
      NCF(5) = NDF
C
C     ----INITIAL GUESS TYPE FOR INTERNAL STATES: CONSTRUCT A LINEAR
C         INITIAL GUESS BETWEEN Y0 AND Y1
C
      INIT(1) = 1
C
C     ----SET INITIAL AND FINAL TIME
C
      Y0(0) = TINITIAL
      Y1(0) = TFINAL
C
C     ----FIX INITIAL AND FINAL TIME
C
      YLB(-1,0) = Y0(0)
      YUB(-1,0) = Y0(0)
      YLB(1,0) = Y1(0)
      YUB(1,0) = Y1(0)
C
C     ----DEFINE GUESSES FOR INITIAL CONDITIONS FOR CONTROL VARIABLES
C
      DO II=NDE+NDAE+1,NDE+NAV
        Y0(II) = 0.D0
      ENDDO
C
C     ----DEFINE GUESSES FOR FINAL CONDITIONS FOR ALL VARIABLES
C
      DO II=1,NDE+NAV
        Y1(II) = Y0(II)
      ENDDO
C
C     ----DEFINE OBSERVATION PATH CONSTRAINT FOR CONTROLS
C
      CALL PTHCON(NTERM,NKON,NCF,IPHASE,ITERM,MXTERM,COEF,
     $    CLB,CUB,CLBL,MXPCON,0.D0,0.D0,1.D0,'ACON1',
     $    'Algebraic constraint 1: u1 = 0',IERPTH)
C
C     ----DEFINE LEAST SQUARES OBJECTIVE
C
      MAXMIN = 2
      CLBL(0) = 'LSQ     DISCRETE DATA'
C
      DO II = 1,NDF
         NTERM = NTERM + 1
C
```

```
C     ----TERM II IS PART OF NLP OBJECTIVE FUNCTION
C
         ITERM(1,NTERM) = 0
C
C     ----TERM II IS COMPUTED IN IPHASE
C
         ITERM(2,NTERM) = IPHASE
C
C     ----TERM II IS ASSIGNED TO DISCRETE DATA
C
         ITERM(3,NTERM) = 2
C
C     ----TERM II IS DISCRETE DATA FUNCTION II
C
         ITERM(4,NTERM) = II
      ENDDO
C
      RETURN
      END
C
C
      SUBROUTINE INIEXP
C
C     ----DATA FOR PARAMETER ESTIMATION PROBLEM
C         PROBLEM DATA IS READ IN FROM FILE TP333.DAT
C
C     ******************************************************************
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
      PARAMETER (MAXRWD=8,MAXCLD=3,NTIME=8)
      COMMON /EXPCM/ STDAT(MAXRWD,MAXCLD),NROWS
      CHARACTER*80 DATFL1
      SAVE DATFL1
      DATFL1='TP333.dat'
C
C     ******************************************************************
C
      IPN = 3
      OPEN(IPN,FILE=DATFL1,STATUS='UNKNOWN')
      READ(IPN,*) NROWS,NCOLS
      IF(NROWS.GT.MAXRWD) THEN
        PRINT *,'NROWS GT MAXRWD; NROWS =',NROWS
        STOP
      ENDIF
      DO II = 1,NROWS
         READ(IPN,*) (STDAT(II,JCOL),JCOL=1,NCOLS)
      ENDDO
      CLOSE(IPN)
C
      RETURN
      END
C
C
      SUBROUTINE EXPDDL(IPHASE,NDD,NDDST,MXDATA,TDATA,DATA,WTDATA,
```

```
     &                  NDATA,IER)
C
C     ----LOAD DISCRETE DATA FOR PARAMETER ESTIMATION PROBLEM
C
C     ***********************************************************************
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
      PARAMETER (MAXRWD=8,MAXCLD=3,NTIME=8)
      COMMON /EXPCM/ STDAT(MAXRWD,MAXCLD),NROWS
      DIMENSION TDATA(NTIME),DATA(NTIME),WTDATA(NTIME)
      PARAMETER (NCONC=1)
      COMMON /CONPAR/ CONC(NCONC)
      COMMON /NDDCNT/ COUNT
C
C     ***********************************************************************
C
C     ----RESET ERROR/SUCCESS CODE TO 0
C
      IER = 0
C
C     ----NUMBER OF DYNAMIC VARIABLES (ODEs and DAEs)
C
      NDYN = 0
C
C     ----LOAD STATE NDD TARGET VALUES
C
      NDATA = NTIME
      SUM_OF_SQRS = 0.D0
C
C     ----INITIALIZE COUNTER FOR NDD FUNCTIONS NOT FITTING A
C         STATE VARIABLE
C
      IF(NDD.EQ.1) THEN
        COUNT = 1
      ENDIF
C
      NDDST = NDD
C
C     ----COMPUTE SUM OF SQUARES SEPARATELY SINCE ALL DATA ARE
C         REQUIRED FOR SCALING = 1
C
      DO II = 1,NTIME
         SUM_OF_SQRS = SUM_OF_SQRS + STDAT(II,2*NDD)**2
      ENDDO
C
C     ----SET TIME VALUES TO FIRST COLUMN IN DATA FILE,
C         CONCENTRATION VALUES TO SECOND (IF ANY), FUNCTION VALUES TO NEXT,
C         AND WEIGHTS TO NEXT. . .
C
      DO II = 1,NTIME
        TDATA(II) = STDAT(II+(IPHASE-1)*NTIME,1)
        DATA(II) = STDAT(II+(IPHASE-1)*NTIME,2*NDD)
        WTDATA(II) = STDAT(II+(IPHASE-1)*NTIME,1+2*NDD)
      ENDDO
```

```
C
C     ----SCALING OF RESIDUALS (IF NEEDED)
C
C           0 - NO ADDITIONAL SCALING
C           1 - DIVISION OF RESIDUALS BY SQUARE ROOT OF SUM
C               OF SQUARES OF CURRENT STATES MEASUREMENTS VALUES
C          -1 - DIVISION OF EACH SINGLE RESIDUAL BY CORRESPONDING
C               ABSOLUTE MEASUREMENT VALUE
C          -2 - DIVISION OF EACH SINGLE RESIDUAL BY CORRESPONDING
C               SQUARED MEASUREMENT VALUE
C
      SCALE = -1
C
      DO II = 1,NTIME
        IF(SCALE.EQ.1) THEN
          IF(SUM_OF_SQRS.NE.0.D0) THEN
            WTDATA(II) = WTDATA(II)*1.D0/SQRT(SUM_OF_SQRS)
          ENDIF
        ELSEIF(SCALE.EQ.-1) THEN
          IF(DATA(II).NE.0.D0) THEN
            WTDATA(II) = WTDATA(II)*1.D0/ABS(DATA(II))
          ENDIF
        ELSEIF(SCALE.EQ.-2) THEN
          IF(DATA(II).NE.0.D0) THEN
            WTDATA(II) = WTDATA(II)*1.D0/(DATA(II)**2)
          ENDIF
        ENDIF
      ENDDO
C
      RETURN
      END
```